DESARROLLO DE HERRAMIENTAS DE GESTIÓN DE PROCESOS PARA GRANJA DE HPC BASADA EN APACHE MESOS

Proyecto fin de grado

Autor: Eva María Expósito Rodríguez

Tutor académico: Harold Yesid Molina-Bulla

Año Académico 2015-2016

Universidad Carlos III de Madrid
www.uc3m.es

Agradecimientos:

Este documento va dedicado a todos los que creyeron en mí en algún momento, y a los que aún lo hacían, cuando ni yo lo hacía. A todos los que están conmigo y a todos los que, por desgracia, ya no pueden estar aquí. A ellos, sobre todo, por cuidarme desde allí arriba y no dejar que me pase nada malo. Este es el final de una etapa y espero que estéis orgullosos.

También merecen una especial mención varias personas. En primer lugar, a mis padres, que me enseñaron el valor de las cosas y que con esfuerzo todo se puede. A mi madrina que me aficionó a la lectura y me abrió nuevos universos. A mis tías y a mis primas por mostrarme que hay un tiempo para dejar de escribir y de leer, y salir de compras, ir al cine, salir de fiesta o simplemente ir a dar un paseo. A mis amigos por soportarme en mis periodos de histeria y cuidarme cuando lo he necesitado. A mi tutor por tener más paciencia que un santo y explicarme cien mil veces las cosas. Y por último a mis abuelos por darme todo el cariño del mundo y defenderme a capa y espada.

Por ultimo quiero agradecer a la gente que no me lo ha puesto fácil. La gente que me hizo daño y que no confiaba en mí. Gracias a ellos soy más fuerte, más paciente y más inteligente y sin ellos no hubiera llegado hasta aquí. Gracias por hacerme ser quien soy.

Universidad Carlos III de Madrid
www.uc3m.es

Índice

Tabla de figuras:

Capitulo 1: Resumen.

En este documento hemos desarrollado como se realiza una librería de **gestión de procesos [4.7]** para un **sistema distribuido [4.2.4.2]** y una interfaz de usuario para la utilización de ésta. Este proyecto viene determinado por la necesidad de encontrar una herramienta para la simulación de tareas en un *entorno multiusuari*o [12.9].

Para determinar qué tipo de computación es la óptima para nuestro problema explicamos los cuatro tipos de computaciones existentes en la actualidad, centrándonos en la computación paralela y distribuida que son las dos más importantes.

Una vez que hemos explicado el tipo de computación elegida explicaremos el sistema distribuido del que disponemos. Nuestro sistema es un conjunto de ***equipos para la computación numérica intensiva (HPC) [4.4,4.4.1,12.2]*** interconectados entre sí. Estos computadores son capaces de paralelizar las tareas en su interior para disminuir el tiempo de procesamiento de éstas.

Para finalizar no podemos realizar un contexto completo de nuestro proyecto si no hablamos de su principal competidor en el sector de la computación a gran escala, los **Supercomputadores [4.5]**. De estos sistemas de procesado realizaremos una visión global, dando los ejemplos más sorprendentes y las ventajas e inconvenientes que tienen frente a los sistemas distribuidos.

Para solucionar los diferentes inconvenientes que tiene la computación distribuida se han generado lo que se conoce como gestor de procesos. Esta herramienta tiene dos partes principales el gestor de colas y el planificador de tareas, cuyo funcionamiento se plantea para finalizar la parte de contextualización del proyecto.

Después de finalizar la contextualización del proyecto en el documento se describen las distintas herramientas que vamos a utilizar para nuestro proyecto.

Por un lado, vamos a utilizar el programa de Apache Mesos. Este software se utiliza de base para crear nuestra librería y para establecer una jerarquía entre las diferentes CPUs que forman nuestro sistema distribuido.

Otra de las herramientas que se explican en este apartado son las diversas estructuras creadas para almacenar los datos que nos proporciona el usuario. Estas son los procesos y las tareas.

Los procesos las estructuras de datos más amplias que tenemos. En ellos se encuentran el identificador del proceso, los recursos necesarios para ejecutar nuestros trabajos y el comando de ejecución de estos.

Por otro lado, están nuestras tareas que son las unidades más pequeñas. En cada una de ellas se guarda un trabajo a ejecutar, los recursos que necesita éste, el identificador propio de la tarea y el nodo al que se envía para procesarla. Estos datos son totalmente necesarios para hacer un seguimiento de los trabajos y que se ejecuten con normalidad en los nodos esclavos.

Además de la arquitectura de nuestro sistema distribuido y del tipo de datos que utilizamos, dentro de las herramientas que utilizamos se encuentra el lenguaje de programación empleado. En el caso de este proyecto nos hemos decantado por Python, debido a su facilidad de aprendizaje y de implementación. Es muy práctico en este caso porque posee programación orientada a objetos lo que nos hace más sencilla la creación de los tipos de datos descritos anteriormente.

En cuanto a la interfaz gráfica que usamos para realizar nuestras simulaciones, hemos optado por el uso de I-Python Books. Esta interfaz es altamente modular y permite simular el código por partes pudiendo tener en un único fichero varios tipos de simulaciones, y no tener que simularlas todas siempre. Esta serie de características la hacen muy práctica para crear una interfaz sencilla.

Por último, hemos desarrollado las distintas funcionalidades de nuestra librería y de nuestra interfaz gráfica, así como las pruebas realizadas para determinar su correcto funcionamiento y sus resultados.

Capitulo 2: Abstract

In this paper we have developed a library of process management and a user interface for a cluster. This project is determined by the need to find a tool for simulating tasks in a multiuser environment.

To determine what type of computation is the best for our problem we explain the four types of computations existing today. But this document is focusing in parallel and distributed computing because they are the two most important today.

Once you have explained the type of distributed computing we explain the chosen system. Our system is a set of interconnected computers HPC. These computers are able to parallelize tasks inside, to reduce the processing time of these.

Finally, we can´t make a good context of our project if we do not speak of its main competitor in the field of large-scale computing, supercomputers. Inside the document, we will take a global view of the processing for these systems, giving the most striking examples and the advantages and disadvantages they have against distributed systems.

To solve the various drawbacks that have been generated in distributed computing exist what is known as process manager. This tool has two main parts the queue manager and scheduler, whose operation is explained for complete the part of contextualization of the project.

After completing the contextualization of the project are described the various tools that we use for our project.

On the one hand, we will use the Apache Mesos program. This software is used as the basis for creating our library and to establish a hierarchy among the different cpus that form our distributed system.

Another tool explained in this section are the various structures created to store the data that the user provides. These are the processes and tasks.

Processes structures are wider data that we have. They have the process ID, the resources needed to execute our jobs and execution commands of these.

On the other hand, they are our tasks are the smallest units. In each one the project run a job saved, the resources you need it, the own identifier of the task and the node that is sent for processing. These data are absolutely necessary to monitor the work and to run normally in the slave nodes.

In addition to the architecture of our distributed system and the type of data we use within the tools we use is the programming language used. For this project we have opted for Python, due to its ease of learning and implementation. It is convenient in this case because it has got OOP what makes us easier to create data types described above.

As for the GUI we use to make our simulations, we have opted for the use of I-Python Books. This interface is highly modular and you can simulate the code parts separately in the same script. This set of features makes it very practical to create a simple and intuitive interface.

Finally, we have developed the different functionalities of our library and our graphical interface, as well as tests to determine their proper operation and its results.

Capitulo 3: Introducción y Motivación

En este documento vamos a desarrollar los procedimientos básicos en la comunicación con una granja de HPC y la realización de un planificador de tareas para poder simular diversas tareas en la **granja de HPC (High performance computation) [12.2,4.4].**

3.1. Motivación

Debido a la necesidad de simular problemas complejos que requieren muchos recursos para temas de investigación en la universidad, se requiere un sistema computacional optimo, en cuanto a recursos, y compatible con multiusuarios. El sistema existente, la granja de HPC del Departamento de Teoría de la Señal y Comunicaciones, soluciona algunos de los problemas, pero sólo sirve para simulaciones muy específicas. Por ello este trabajo va orientado a crear un planificador de tareas compatible con los sistemas existentes y que pueda realizar cualquier tipo de simulación.

3.2. Objetivos Generales

Los principales objetivos se este proyecto son la creación de una librería que permita la ejecución de múltiples ficheros en un entorno de HPC y una interfaz intuitiva para los usuarios, para evitar el uso de la línea de comandos. Para ello se deben realizar tareas con los diversos ejecutables y definir el estado de éstas en todo momento. Los diferentes ejecutables se introducirán por el usuario mediante un *script (12.1)* en lenguaje Python. Para realizar estos procesos nuestra librería se basará en la *API* (interfaz de operación de aplicaciones) de Apache mensos.

3.3. Objetivos Específicos

Dentro de los objetivos específicos de este proyecto tenemos que destacar la compatibilidad con un entorno preestablecido, la granja de HPC del departamento de TSC, que posee un número determinado de recursos.

Por otro lado, el software a desarrollar debe que ser compatible con otros gestores de recursos, existentes en los equipos.

Otro de los requisitos indispensables es la minimización de los cambios en la estructura existente, y en el software disponible en las diversas máquinas que componen este entorno. Esto se debe a que el entorno de ejecución está en uso y cualquier cambio podría suponer una pérdida de información y de recursos.

Una facilidad que se le añade al proyecto es que las tareas pertenecientes a un proceso común necesitan la misma cantidad de recursos y son independientes entre ellas. Es decir, los resultados de una tarea no dependen de la ejecución de la otra tarea.

Y, por último, se debe utilizar un sistema de planificación compatible con los existentes en el departamento. En la actualidad el departamento hace uso de Apache Spark para el cómputo de procesos de *Big Data*.

Capitulo 4: Estado del arte

En este apartado procederemos a la contextualización del trabajo. Veremos una visión general de la computación de tareas y haremos un breve recorrido por los diversos tipos existentes, centrándonos en la computación distribuida y sus ventajas frente a otro tipo de computaciones.

4.1. ¿Qué es la computación?

Existen varias definiciones de computación. Pero todas ellas siempre vienen asociadas a la definición de la palabra informática. Ambas palabras se encuentran inter-relacionadas, pero no debemos olvidar que son diferentes. Por ello en el siguiente sub-apartado nos dispondremos a explicarlas y a mostrar las similitudes que nos llevan a la confusión que se produce cada vez que hablamos de ellas.

4.1.1. Diferencias entre computación e informática

Para identificar las diferencias entre ambas palabras en primer lugar nos debemos disponer a definirlas.

> ***"La computación** es la ciencia **encargada de estudiar los sistemas**, más precisamente computadoras**, que automáticamente gestionan información." [11.1.5]***
>
> *"La **informática** se refiere al **procesamiento automático de información** mediante **dispositivos electrónicos** y **sistemas computacionales**." **[11.1.6]***

De estas definiciones podemos deducir que la informática usa la computación para realizar el procesado de datos de manera automática, haciendo que la computación, sea una especialidad de la informática.

Una vez tenemos clara la definición de computación vamos a pasar a explicar los diferentes tipos existentes y cada una de sus características.

4.2. Tipos de computación

Existen 4 tipos de tipos de procesamiento de tareas dependiendo del número de equipos y de cómo se distribuyan los procesos en ellos. En los siguientes sub-apartados se detallarán diversos aspectos de los tipos de computación para al final elegir los más adecuados para nuestro proyecto.

4.2.1. Computación monolítica

En primer lugar, procederemos a definir la más simple de las computaciones, y la más antigua. Siendo ésta, la que se empleó en las primeras máquinas de procesado de datos.

4.2.1.1. Definición

"La **computación monolítica** se lleva a cabo cuando, en el procesado de información, sólo interviene una única CPU *[12.5]."* ***[11.1.4]***

Éste método conlleva a un fácil seguimiento de la tarea de ejecución, así como su estado y otras características que comentaremos en el apartado posterior.

Ilustración 1 Computación monolítica[11.3.6]

4.2.1.2. Características generales

Esta definición, nos lleva a determinar que, los recursos son limitados a los que contiene la unidad de procesamiento del sistema monolítico.

Por otro lado, la comunicación con este tipo de sistema puede ser multiusuario o monousuario. Dependiendo de esto las características del acceso de datos y su procesamiento cambian.

Si es monousuario, o de usuario único, éste dispone de toda la capacidad de cómputo del equipo para realizar el procesado de la información. Esto nos proporciona la ventaja de que no existe desaprovechamiento de recursos en nuestra red.

En cambio, si se trata de un sistema multiusuario, la distribución de los recursos se realiza mediante la división en tiempo de los recursos del equipo. Es decir, cada usuario dispone de la totalidad de recursos, en un tiempo limitado, para la ejecución de sus tareas, quedando éstas en pausa, mientras se ejecutan las tareas de otros usuarios. Éstas características, hacen que el sistema no sea muy práctico, cuando el número de usuarios es muy amplio y que programas simples tarden mucho tiempo.

Este sistema es un sistema muy simple que está cayendo en desuso debido a la limitación de recursos que tiene y a sus problemas con los sistemas multiusuario, que son los más utilizados en la actualidad.

4.2.2. Computación paralela

En este apartado, veremos el procesamiento de datos más utilizado en la actualidad, dado que, la mayoría de ordenadores personales, y demás dispositivos electrónicos, poseen varios núcleos de procesamiento.

4.2.2.1. Definición

> "La **computación paralela** es un sistema de procesado de tareas que se realiza utilizando multiproceso coordinado mediante un sistema de memoria compartida." **[11.1.4]**

Para poder comprender bien esta definición debemos realizar una definición del un sistema multiproceso. Pero este sistema es muy complejo para realizar una definición directa sin realizar previamente la definición de un sistema multitarea.

Los sistemas multitarea son sistemas que utilizan la división en tiempo de sus recursos para la ejecución de varios **procesos mono-hilio, un proceso milti-hilo** o la combinación de ambos considerando una tarea a cada hilo de ejecución**.**

Teniendo esta definición clara podemos definir un sistema multiproceso como un sistema de dos o más procesadores en los que se ejecuta un sistema multitarea en paralelo, pudiendo ejecutarse una tarea en varios procesadores de manera simultánea.

Una vez tenemos clara la definición de multiproceso vamos a recuperar la definición de computación paralela.

Según la definición la computación paralela es un sistema de procesado de tareas que realiza la separación de los diferentes hilos de procesado entre

diversas CPUs (unidades de procesamiento) dividiendo los recursos de éstas en tiempo y dedicando un intervalo de tiempo al procesamiento de una parte pequeña del hilo. La información que se obtiene de procesar se guarda y en caso de ser necesaria por otro CPU, se intercambia a través de una red de memoria compartida.

Ilustración 2 Computación paralela[11.3.5]

En la ilustración superior podemos observar una máquina de computación paralela por dentro pero no siempre la computación paralela se realiza en una única máquina debido al elevado coste de fabricación que tienen este tipo de equipos. Por esta característica existen dos tipos de computación paralela.

La computación paralela multiproceso, es la que se implementa en una única máquina y los procesos de compartición de información se hacen mediante la memoria compartida interna del dispositivo.

La computación paralela distribuida se basa en el mismo sistema, pero las diferentes unidades de procesamiento son máquinas independientes. Esto hace que necesitemos una red entre los diversos equipos que realice las funciones de la memoria compartida. Esta red determina el tipo de calidad que tiene el sistema de computación paralela distribuida.

4.2.2.2. Características generales

Su principal inconveniente es la dificultad para gestionar errores y la dependencia entre subprocesos de las tareas para realizar las ejecuciones. Esto se debe a la dependencia entre las sub-tareas y los procesos. Es decir, si una sub-tarea no finaliza correctamente el proceso se detiene.

Además, tenemos el problema de la gestión de recursos de las diferentes unidades de procesamiento automática, que desperdiciarán recursos del dispositivo dependiendo de los criterios de reparto de éstos.

Las ventajas vienen de la mano de la velocidad de procesado de las tareas, siendo la velocidad de procesado mayor conforme se añaden unidades de procesamiento.

Por otro lado, el avance en la creación de dispositivos de procesamiento cada vez más eficientes y más pequeños ha contribuido a su rápida expansión. Esta gran ventaja hace que sea el sistema más usado en la actualidad para todos los dispositivos orientados a consumidores. Los computadores personales, los dispositivos móviles y la mayoría de elementos que usamos en nuestro día a día.

Otro tipo de dispositivos que usan este sistema son los denominados supercomputadores de los que hablaremos más adelante que realizan su computación de manera paralela de manera masiva. Es decir, tienen a su disposición millones de unidades de procesamiento para paralelizar sus procesos. En estos equipos se usan ambos tipos de computación predominando en la actualidad la computación paralela distribuida por la minimización de costes de fabricación de los sistemas.

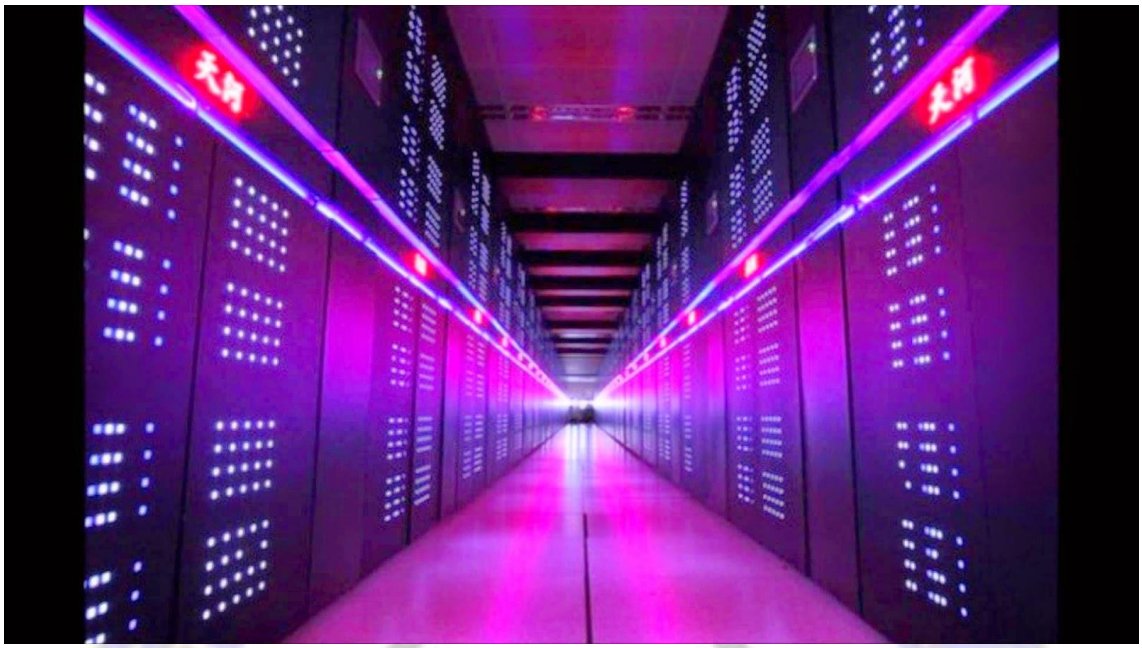

Ilustración 3 Tianhe-2. Segundo supercomputador más potente del mundo según el top 500 [11.3.6]

4.2.3. Computación cooperativa

Es la más inusual de las computaciones existentes. En el siguiente sub-apartado veremos, tanto su explicación, como sus diferentes usos.

4.2.3.1. Definición

La computación cooperativa consiste en la utilización de varias unidades de procesamiento, igual que en las computaciones distribuida y parte de la paralela, pero con las diferentes máquinas en diversos puntos del mundo. Esto se denomina Grid computing.

4.2.3.2. Grid computing

"La **computación en rejilla** combina los ordenadores de múltiples sitios para alcanzar un objetivo común, resolver una sola tarea, y luego puede desaparecer con la misma rapidez." **[11.1.10]**

Una de las principales estrategias de la computación grid es el uso de middleware para dividir y distribuir tareas de un programa de entre varios ordenadores, a veces hasta muchos miles de ellos. Grid computing implica la computación de una manera distribuida, y no el uso de un sistema distribuido, que también puede implicar la agregación de grupos a gran escala. Lo que nos llevaría a la computación cooperativa. Que utiliza un sistema de rejilla a gran escala con los equipos que realizan el procesado en diversas partes del mundo**. [11.1.10]**

4.2.3.3. Usos

Diferentes usuarios ponen a disposición de una causa, los excedentes de recursos de las unidades de procesamiento que su ordenador no utiliza para ejecutar los proyectos que utilizan ese tipo de computación.

Uno de los proyectos más significativos, el proyecto *SETI (Search for ExtraTerrestrial Intelligence)*, que se están ejecutando es para la búsqueda de vida extraterrestre. Millones de ordenadores eje en el mundo ejecutan parte de un algoritmo informático, dedicado a investigar mediante análisis de señales electromagnéticas capturadas en distintos radiotelescopios si alguna de ellas proviene de vida extraterrestre.

Los principales inconvenientes de este tipo de computación son las diferentes disponibilidades de recursos y la variabilidad de éstos en diferentes instantes de tiempo.

4.2.4. Computación distribuida

A continuación, definiremos el tipo de computación elegida en este caso por las características de nuestro tipo de trabajo, y el tipo de instalaciones físicas del que disponemos.

4.2.4.1. Definición

> "La **computación distribuida** es la ejecución de **diversos procesos** o tareas en un **sistema distribuido**." **[11.1.4]**

Según esta definición tenemos varios problemas para comprenderla en su totalidad. En primer lugar, necesitamos averiguar que son los procesos y las tareas, que se explicarán en los apartados 5.2.1 y 5.2.2 de este documento. Por otro lado, necesitamos saber que es un sistema distribuido. A continuación, haremos una breve explicación de este tipo de sistemas.

4.2.4.2. Sistemas distribuidos

Existen varias definiciones de sistemas distribuidos, pero la que mejor se adapta en el contexto de este documento es la realizada por G. Colouris, J. Dollimore, T. Kindberg en el libro **Sistemas distribuidos: Conceptos y diseño**, en el que se especifica que:

"Un sistema distribuido es aquel en el que los componentes localizados en computadores, conectados en red, comunican y coordinan sus acciones mediante el paso de mensajes." ***[11.1.1]***

Ilustración 4 Granja de HPC(High performance computation) de servidores de Apple[11.3.4]

Por lo tanto, con la definición de computación distribuida explicada anteriormente podemos determinar que la computación distribuida consiste en la utilización de varios componentes por uno o varios usuarios para realizar la ejecución de una o varias tareas.

Debido a que es la solución usada para nuestro objetivo general, a continuación, vamos a detallarla con mayor claridad, a explicar sus diversas cualidades y a ilustrar varios usos existentes en la actualidad.

4.3. Computación distribuida

Una vez tenemos claro cuál es el concepto de computación distribuida, vamos a desarrollar sus principales características y a explicar con detalle varios ejemplos de sistemas que usan computación distribuida. Más adelante expresaremos las ventajas y las desventajas de este tipo de redes, tomando como ejemplo las granjas de HPC **[4.5.1].**

4.3.1. Características principales

La principal característica de este tipo de procesado es la paralelización de las tareas de diferentes usuarios, evitando los tiempos de espera. Esto se debe, a que la asignación de recursos no es total a un único usuario, sino que, se dividen los recursos, dependiendo de los recursos solicitados para cada uno. Este mecanismo puede dar problemas de *Fairness [12.3]*, es decir, puede llegar un usuario y pedir la totalidad o una gran parte de los recursos. De esa manera el sistema quedaría colapsado y los usuarios que quisieran acceder no podrían. Este problema se solucionaría imponiendo sistemas de gestión de colas que den diversas prioridades a los proyectos.

Otra característica de este tipo de computación es la escalabilidad de su arquitectura. Al no poseer una arquitectura fija es muy fácil añadir nuevas máquinas a la red y realizar los cambios de software locales. En cambio, cuando los cambios de software son generales de todos los nodos es más complejo realizarlos.

4.3.2. Ejemplos

En la actualidad existen muchos ejemplos de redes distribuidas de comunicaciones. La más representativa de ellas es una red que usamos cotidianamente, Internet.

Internet posee una red enorme, con multitud de ordenadores, que se conectan mediante mensajes HTTP. Este tipo de red es abierta, lo que hace que crezca más rápidamente. Este tipo de conexión es la más usada en la actualidad dada su universalidad y la creación de nuevas tecnologías que la actualizan y la ayudan a mejorar.

Ilustración 5 Esquema del funcionamiento de internet [11.3.3]

Otro ejemplo de redes distribuidas son las intranets de las empresas, universidades etc. Estas redes son privadas y poseen mayores mecanismos de seguridad, pero el principio es el mismo que el de internet.

4.4. Granjas de HPC

Para realizar nuestro proyecto disponemos de lo que comúnmente se llama granjas de HPC. En este contexto la palabra granja hace alusión a conjunto de ordenadores o red de ordenadores, dejando su significado más usual apartado. Teniendo este concepto claro, vamos a comprender qué tipo de computadores son los HPC y cuáles son sus principales ventajas.

4.4.1. Que son los HPC

Los HPC o computación de alto rendimiento son equipos que realizan el procesado de datos mediante **computación paralela distribuida** [4.2.2], por la cual, los equipos dividen las tareas en sub-tareas encargando a cada uno de sus procesadores una mediante el uso *hilos (threads) [12.4]*. Estos equipos se distribuyen en un sistema, para poder formar la granja de ordenadores o granja de HPC (High performance computation). De esa manera, cogemos las ventajas de la computación paralela de procesos y las características de la computación distribuida para solucionar los problemas del multiusuario.

4.5. Supercomputadores

Las supercomputadoras son el tipo de ordenadores más potentes y más rápidos que existen en un momento dado. Estos equipos pueden procesar enormes cantidades de información en poco tiempo, pudiendo ejecutar millones de instrucciones por segundo. Están destinadas a una tarea específica y poseen una capacidad de almacenamiento muy grande. Son unos dispositivos de procesamiento muy caros y con grandes gastos tanto de fabricación, como de mantenimiento, debido a la gran cantidad de energía que consumen.

Además, necesitan un control de temperatura especial para poder disipar el calor que algunos de sus componentes pueden llegar a alcanzar.

Estos dispositivos controlan el acceso a todos los archivos y deciden la distribución de los recursos a las diferentes tareas. También controla las operaciones de entrada y salida. El usuario se dirige a la computadora central del superordenador cuando necesita procesar información o consultarla.

Estos equipos están diseñados para sistemas de multiprocesamiento, es decir, pueden soportar a miles de usuarios en línea.

Por último, cabe destacar que la cantidad de procesadores que puede llegar a tener estos mecanismos, depende principalmente de las características de ellos, que mejoran según avanza el mercado. En el siguiente sub-apartado podremos observar cómo varían dependiendo del modelo y cuáles son las diferentes tasas de procesado.

4.5.1. Supercomputadores en el Mundo

En la actualidad, existen muchos superordenadores a pesar de su coste y el coste de mantenimiento. Estos son usados para la seguridad cibernética de diversos países y como sistemas de computación ultra-avanzados en cuestiones de investigación. En la siguiente imagen tenemos una captura de pantalla con los 5 superordenadores más potentes de la actualidad y su localización, incluyendo en ella las principales características del sistema.

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,659.9

Ilustración 6 Top 5 de los supercomputadores más potentes. Junio 2016 [11.3.1]

Esta lista ha sido obtenida de la página web de Top 500, que es un proyecto creado para realizar un ranking de las 500 supercomputadoras con mayor rendimiento del mundo. La lista ha sido recopilada por Hans Meuer de la Universidad de Mannheim (Alemania), Horst Simon y Erich Strohmaier, del NERSC y del Lawrence Berkeley National Laboratory, y por Jack Dongarra de la Universidad de Tennessee (Knoxville). Según la última lista publicada (Junio 2016) el super-computador más potente del mundo es el que se encuentra en el centro nacional de computación de wuxi – china. **[11.3.1, 11.1.8]**

Ilustración 7 Sunway TaihuLight [11.3.2]

Este superordenador está siendo usado para investigación. Como se muestra en la imagen posterior el Sunway TaihuLight posee una tasa de computación real de 93014,6 TFlop/s, consume una potencia de 15371 kW. El procesador es de 1,45 G HZ y tiene 1310720 GB de memoria para realizar las operaciones.

Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway

Site:	National Supercomputing Center in Wuxi
Manufacturer:	NRCPC
Cores:	10,649,600
Linpack Performance (Rmax)	93,014.6 TFlop/s
Theoretical Peak (Rpeak)	125,436 TFlop/s
Nmax	12,288,000
Power:	15,371.00 kW
Memory:	1,310,720 GB
Processor:	Sunway SW26010 260C 1.45GHz
Interconnect:	Sunway
Operating System:	Sunway RaiseOS 2.0.5

Ilustración 8Caracteristicas del Sunway TaihuLight [11.3.1]

4.6. Ventajas e inconvenientes de la Granja HPC frente a los Supercomputadores

En este apartado vamos a describir las principales ventajas e inconvenientes de una granja de HPC (High performance computation) frente a los Supercomputadores.

4.6.1. Ventajas

La principal ventaja de una granja de computadoras frente a un supercomputador es la disminución de gastos. Tanto el coste de fabricación como el de mantenimiento de una granja HPC son muy inferiores con respecto a los superordenadores.

Otra ventaja de una granja HPC es su alta modularidad y la facilidad de añadir y quitar equipos de ella. Eso nos proporciona una gran escalabilidad en el sistema, que los supercomputadores no poseen.

Para finalizar, otra ventaja de estos sistemas frente a los superordenadores, es su alta disponibilidad. La disponibilidad de recursos es mayor debido a que la petición de recursos la realizan los usuarios. En cambio, en un supercomputador, si éste dispone de recursos los asignará todos a los procesos activos, por lo que si existen varios procesos complejos ejecutando la disponibilidad de recursos será limitada.

4.6.2. Inconvenientes

Existen varios inconvenientes de los sistemas distribuidos frente a los superordenadores. La primera de ellas consiste en que, los supercomputadores poseen mayor capacidad tanto de cómputo como de memoria en comparación con las granjas de ordenadores.

Además, poseen una distribución paralela de tareas que hace que no se produzcan bloqueos a la hora de realizar cómputos de grandes dimensiones a la vez que pequeñas tareas independientes.

Otro inconveniente de este sistema es la velocidad de la comunicación entre diferentes tareas que en el caso de supercomputadores es mucho más amplia que en los sistemas distribuidos por la necesidad de pasar por el nodo maestro o líder de nuestro sistema, mientras que en los supercomputadores existen zonas de memoria compartida que agilizan este proceso.

El último inconveniente a destacar son los problemas de disponibilidad de recursos en las granjas de computadores.

Para resolver los problemas anteriores se han creado herramientas de software. Estas herramientas son los gestores de procesos. Su funcionamiento es fácil y varían en dificultad según el tipo de gestor elegido.

4.7. Gestores de procesos para computación distribuida

A continuación, vamos a definir lo que son los gestores de procesos de una granja de HPC (High performance computation) y cuál es su funcionamiento. Además de realizar una breve visión de varios gestores existentes.

4.7.1. Definición

Los sistemas de gestión de procesos, gestionan una cola de ejecución, planifican la ejecución de las tareas y gestionan los recursos, para minimizar costes y maximizar rendimiento de las aplicaciones. Estos sistemas han sido creados para resolver los inconvenientes de las granjas de HPC es realizar una repartición de recursos justa. Tienen dos partes diferenciadas.

El gestor de colas realiza un control en la petición de recursos al sistema por parte de cada usuario. Este sistema se ocupa del problema del *fairness [12.3]* y evita que un único usuario acapare todos los recursos de la granja de HPC. En este trabajo el gestor de colas utilizado es el existente en nuestro sistema por defecto. Este gestor asigna los recursos a un usuario siempre que haya recursos disponibles, lo que no resuelve el problema. En este caso es óptimo debido a que la carga de procesado no es muy amplia al igual que el número de usuarios del sistema.

El planificador de tareas realiza la asignación de tareas a los diferentes nodos de la granja de ordenadores, según los requisitos necesarios de memoria y de procesamiento impuestos por el usuario final. Además, hace el seguimiento de los procesos dentro del sistema de ejecución, devolviendo los errores de las diversas tareas y el estado de finalización. En este proyecto en particular estos datos se pueden guardar en un fichero para consultarlos posteriormente, así como los nodos a los que ha sido asignada cada tarea.

4.7.2. Funcionamiento

El funcionamiento básico de un gestor de recursos tiene tres partes básicas.

La primera es el registro. Los usuarios envían trabajos indicando requisitos de memoria, tiempo de procesador y espacio en disco. El gestor de recursos registra el trabajo y se comunica con los diversos nodos de la arquitectura en busca de recursos disponibles.

La segunda parte es la asignación y procesado de las tareas. Esta se realiza tan pronto los recursos pedidos se hallen disponibles. El planificador pone a ejecución el trabajo solicitado en los nodos asignados por el gestor y asigna las tareas a los nodos elegidos por él.

La última parte es la recogida de resultados. Para ello el *Scheduler [12.6]* consulta el estado de los trabajos, que en este caso son tres: en ejecución, en espera o terminados. En caso de que todas las tareas hayan terminado este mecanismo devuelve el estado de finalización del proceso y cierra la conexión con el sistema distribuido.

4.7.3. Gestores de procesos existentes

Existen varios tipos de gestores de procesos en la actualidad. Los gestores que pueden compilar múltiples tipos de datos más destacados en la actualidad son Condor y SGE.

Además de estos dos casos, en este apartado vamos a comentar el otro gestor de tareas existente en el sistema distribuido del departamento, Apache Spark.

4.7.3.1. Sun Grid Engine (SGE, o Oracle Grid Engine)

Es un gestor de colas de Oracle de código abierto. Su funcionamiento básico sigue los parámetros descritos anteriormente, añadiendo prioridades con un sistema de tickets para evitar los casos de bloqueo. Es decir, posee un gestor de colas interno.

4.7.3.1.1. Funcionamiento

Antes de comenzar a detallar el funcionamiento es necesario explicar algunos conceptos, como job y cola. El job es una unidad de trabajo que es definida por un script. Es equivalente a la tarea en nuestro sistema. La cola: es un contenedor para una categoría de trabajos que pueden ser asignados para ser ejecutados en un único CPU.

Los usuarios envían trabajos (jobs) especificando el perfil de requisitos (CPU disponible, disco, arquitectura...). SGE registra el trabajo, su perfil de requisitos e información de control (usuario, grupo, proyecto, departamento, fecha/hora de envío...). Tan pronto como una cola, quede disponible, SGE lanza a ejecución uno de los trabajos en espera:

Aquel con mayor prioridad o mayor tiempo en espera, según la configuración del planificador. En caso de que haya varias colas disponibles se escoge la más libre. Puede haber varias colas por cada equipo debido a que las colas se corresponden con un tipo de trabajo que solo puede realizar un tipo de CPU.

A partir de estos datos ya podemos explicar el último punto del funcionamiento del SGE, conocido como ciclo de vida del trabajo o job, donde se observa el camino que sigue cada trabajo desde que un usuario lo envía hasta que recibe el resultado. El ciclo de vida de un trabajo es el siguiente:

En primer lugar, se envía el trabajo al host maestro. Éste almacena el trabajo en su base de datos. Cuando hay una cola disponible, el equipo maestro asigna le el trabajo y lo envía la unidad de procesamiento correspondiente. El equipo de ejecución guarda el trabajo en su base de datos, lo inicia y se mantiene a la espera hasta obtener el resultado. Cuando ha termina envía el resultado al equipo maestro y elimina el trabajo de su base de datos.

4.7.3.2. Condor

Condor ofrece un mecanismo de trabajo en cola, una política de planificación, un esquema de prioridades, el seguimiento de los recursos y la gestión de estos. Los usuarios envían sus puestos de trabajo en serie o paralelos, a Condor, y este los sitúa en una cola, decide cuándo y dónde ejecutar los trabajos en base a una política propia, monitorea cuidadosamente su progreso, y en última instancia, informa al usuario sobre la terminación de éste.

4.7.3.3. Apache Spark

Es un planificador de tareas para procesos de Big data que necesitan una gran capacidad de cómputo. Se desarrolla en entornos de granja de HPC que dispongan de una distribución del Software libre de Apache. Es compatible con diversos entornos clúster y sistemas, tanto, Unix como Windows.

Este software usa un tipo de fichero predeterminado para Apache Spark que sólo es compatible con Apache Spark en la salida de datos. Este inconveniente hace necesaria la implementación de nuestro gestor de recursos básicos para nuestro sistema de computación distribuido.

Capitulo 5: Herramientas de la solución

En este apartado del documento, se desarrollarán las diferentes herramientas utilizadas para la creación de nuestro gestor de procesos. En primer lugar, desarrollaremos la arquitectura interna de la granja de HPC del que disponemos. Y, en segundo lugar, se desarrollará una explicación del lenguaje de programación utilizado para desarrollarlo, las ventajas que nos llevaron a elegirlo y la interfaz de usuario de la que disponemos.

5.1. Apache Mesos

Apache Mesos es un tipo de software que distribuye los elementos de una granja de HPC (High performance computation) y realiza la interconexión de éstos. Esta herramienta es software libre, es decir, no posee derechos de autor, y está implementada para que sea compatible con varios lenguajes de programación.

En el siguiente apartado se describirá la estructura básica de nodos determinada por esta herramienta y se determinarán las diferentes funciones de éstos y los tipos de interconexión que poseen entre ellos.

5.1.1. Arquitectura básica

Como ya he explicado anteriormente, las granjas de ordenadores se basan en la distribución de procesos, en los diferentes nodos de la red. En esta arquitectura, existen varios tipos de nodos, representados en la figura posterior, en la que también se muestran las diferentes conexiones entre ellos.

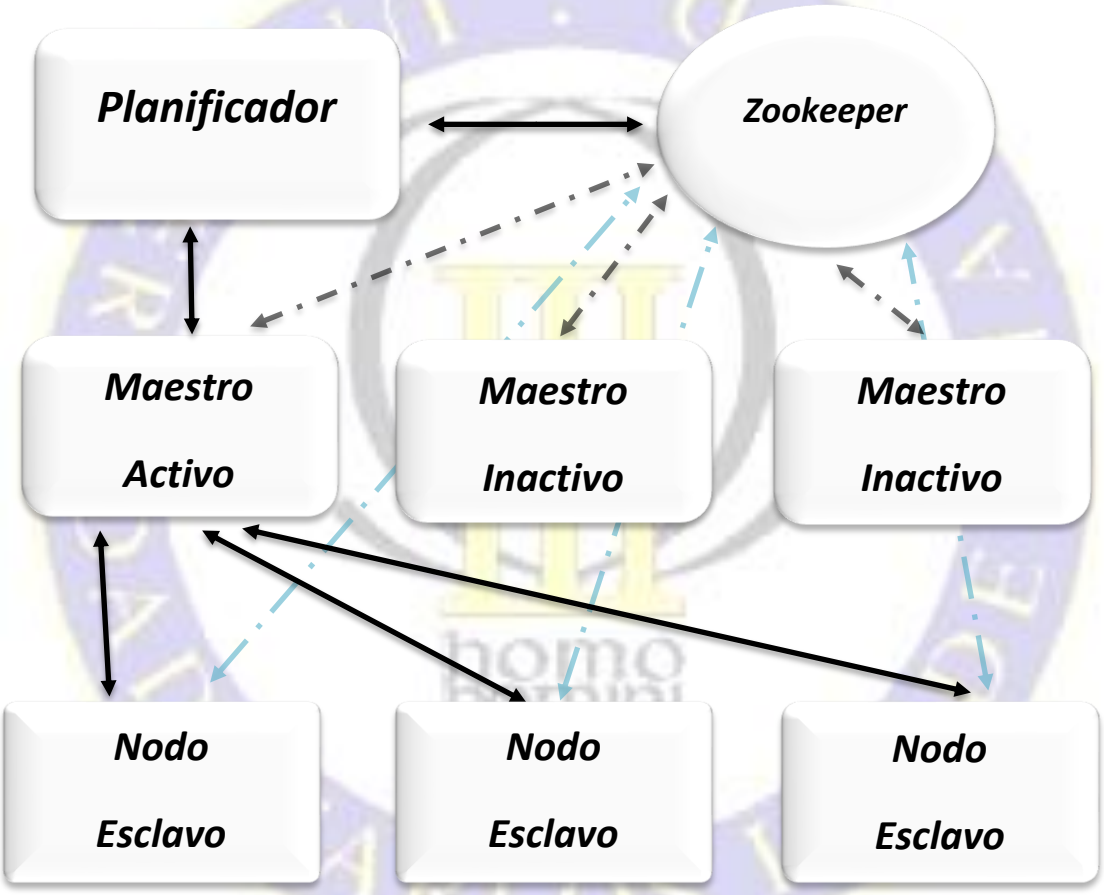

Ilustración 9 Arquitectura básica Mesos

Una vez tenemos el esquema mental de esta distribución de nodos, procederemos a definir cada uno de ellos, determinando sus principales funciones en la ejecución de las tareas.

5.1.2. Scheduler o planificador

En primer lugar, vamos a explicar el funcionamiento del planificador de tareas. Este elemento es un nodo virtual, es decir no existe físicamente y se encarga de generar los procesos y las tareas que conforman cada uno de ellos. Es decir, es una interfaz en la que los diferentes usuarios definen los diferentes trabajos que componen cada proceso y la cantidad de recursos necesarios por ellos para su ejecución. Una vez el planificador ha realizado la constitución de éstos elementos se realizará la petición de recursos y la asignación de estos mediante varios pasos que serán explicados en apartados posteriores.

5.1.3. Zookeeper

El Zookeeper es un supervisor de recursos de alta disponibilidad. Su principal función consiste en la búsqueda del nodo maestro en nuestro sistema. Esta función se debe a que una granja de HPC determinada posee más de un nodo maestro, por si ocurre cualquier problema en las diferentes conexiones que poseen.

El Zookeeper posee una lista de nodos maestros posibles y realiza peticiones de conexión con cada uno de ellos para determinar cuál es el nodo maestro activo. Este mecanismo nos evita tener una lista de nodos maestros, en cada uno de los usuarios finales de la granja de HPC y en cada uno de los nodos agentes, y simplifica los procesos de escalabilidad cuando se quiere añadir o eliminar un nodo maestro.

5.1.4. Máster o Nodo Maestro

Es el elemento más importante de la arquitectura. En nuestra red disponemos de tres másteres. Este nodo físico se encarga de la conexión del planificador con los nodos esclavos, que van a ejecutar las tareas. Aunque sólo existe un nodo Maestro activo. Los nodos maestros inactivos se utilizan de salvaguarda por si el nodo Maestro principal sufre errores o algún tipo de Bloqueo (fallos de conexión, pérdida de corriente, sobrecalentamiento, etc.). Para que un nodo Maestro se active el resto de nodos Maestros debe de estar inhabilitados.

Estos nodos de encargan de la comunicación del panificador de tareas con los nodos agentes. Realizan la petición de recursos y el envío de las tareas a las CPU que han sido asignadas por el planificador.

5.1.5. Nodo Esclavo o agente

El nodo esclavo es el encargado del procesado de las tareas. El nodo esclavo envía sus recursos al nodo maestro. Cuando el nodo maestro envía una tarea al nodo esclavo, éste reserva los recursos y empieza la ejecución de ésta. Cuando la tarea finaliza el nodo agente envía el mensaje con el estado de la tarea (finished, killed o failed), dependiendo de si la tarea finalizó correctamente, fue terminada abruptamente o falló. Si el estado es Failed también envía el error ocurrido para identificar el motivo al nodo maestro. Una vez la tarea ha finalizado el nodo esclavo la elimina de su memoria interna.

5.2. Elementos de ejecución

En este apartado vamos a distinguir como está distribuida la parte software del proyecto, es decir, vamos a introducir cuales son las principales estructuras que usamos para ejecutar los diversos Scripts proporcionados por el usuario.

5.2.1. Procesos

Los procesos son un conjunto de elementos de ejecución. Los programas informáticos dividen todas las “tareas” en grupos. Éstos están formados por elementos de similares características para que los recursos de ejecución que necesiten sean iguales. Esta estructura se denomina proceso. Este sistema facilita la ejecución de cálculos complejos.

En el caso que ilustramos en este documento, su estructura interna además posee varios tipos de datos, usados para optimizar las conexiones. Uno de ellos es el identificador del proceso que, en este caso, es el identificador de la conexión con la granja de HPC. Este tipo de dato es un número, que sirve para hacer un control del proceso, y asignar los recursos necesarios. Por otro lado, posee un elemento con el número de recursos que necesita cada tarea. En nuestro caso son dos variables, el número de CPUs (nodos esclavos) dedicadas a cada tarea, y la cantidad de memoria asignada de cada unidad de procesamiento. El último tipo de datos que incorpora es el número de tareas a ejecutar en él.

5.2.2. Tareas

Es la menor de las estructuras de datos de este proyecto y la más simple. Se genera en el interior del planificador. Y se utiliza la generada en la API de Apache Mesos. Entre las variables que contiene usaremos 5 principalmente. A continuación, nos dispondremos a detallarlas, tanto el tipo de datos que son, como sus funcionalidades en cada tarea.

La primera de ellas es el identificador de la tarea que sirve para ver el estado de ésta, en nuestro código aparece como el nombre de id_value.

La segunda es el comando a ejecutar en la tarea. Éste es un comando para ejecutar el fichero Python del usuario que corresponde a un trabajo. Este comando tiene la forma 'exec (directorio del fichero)' como se puede comprobar en el siguiente ejemplo

'exec /export/usuarios01/eexposito/Datos/run_test'

La tercera variable contiene el nombre de la tarea. En nuestro caso las tareas se llaman Tarea y el número del identificador.

El cuarto tipo de datos usado es el valor del identificador del esclavo(task.slave_id.value). Como se puede deducir por el nombre, este es el sitio en el que se guarda el identificador del nodo agente al que se le ha asignado la tarea para su ejecución.

El quinto tipo de datos guarda los recursos asignados a cada tarea en su interior. Éste tipo de datos es un objeto en el que se añaden dos objetos más, el objeto CPUs y el objeto mem, a estos objetos se les especifica un nombre, el tipo de datos que son y el valor que proporcionó el usuario para el tipo de datos, es decir, si son los CPUs, en este apartado se le añade el número de unidades de procesamiento que el usuario quiere que se le asigne a cada tarea.

5.3. Python

En este apartado vamos a realizar una breve explicación del lenguaje de programación utilizado. Además, veremos una breve introducción de la interfaz de usuario usada para ejecutar nuestro proyecto, y las diferentes ventajas que nos proporciona ésta.

5.3.1. Introducción

En este proyecto se utilizará Python 2.7 para su implementación. Aunque ya han aparecido versiones posteriores, esta versión es compatible con nuestro sistema distribuido y al ser una versión consolidada posee menos errores. El principal inconveniente de esta elección, es la posibilidad de que se quede obsoleto antes, pero es un problema despreciable, porque se puede seguir utilizando la versión y en un futuro siempre se puede actualizar ésta.

5.3.2. Ventajas de uso

Las principales ventajas de uso de este lenguaje son su fácil aprendizaje, y su capacidad de usar programación orientada a objetos. La facilidad de

aprendizaje está dada por su lenguaje intuitivo y por la similitud con otros lenguajes más conocidos como C o Java. Al igual que java posee programación orientada a objetos, lo que facilita la encapsulación de la información y la utilización de ella de una manera práctica más fácil de entender por el usuario.

5.3.3. Tipos básicos utilizados

Los tipos básicos de datos en Python son los mismos que existen en C. Pero estos tipos poseen una particularidad, no es necesario asignar un tipo de datos hasta que no posea un valor determinado, y entonces se asigna por defecto.

Por otro lado, en este proyecto vamos a utilizar los objetos de Python, tanto unos que crea Mesos por defecto para comunicarse con el sistema distribuido, como nuestro objeto planificador, que crea las diferentes conexiones y guarda el estado de ellas. Este objeto es utilizado por el fichero de pruebas para poder realizar diversas simulaciones.

Además, realizaremos operaciones de entrada y salida en diversos ficheros para guardar los estados de los procesos y los posibles errores generados.

5.3.4. I-PythonBook (IPYNB)

El I-PythonBook es una interfaz gráfica diseñada para la creación y ejecución de código en Python. El texto codificado puede dividirse en celdas para realizar diferentes pruebas. Esta plataforma te permite simular tu código y te devuelve los diferentes errores de Python por pantalla. Es bastante práctica porque se puede utilizar mediante interfaz web en el puerto que tú desees.

En el proyecto es utilizado para la ejecución de las diversas pruebas debido a su modularidad y a que no necesitar ejecutar el fichero entero cada vez que realizas una simulación. Lo que permite realizar las diversas pruebas por separado. Estas características hacen a este programa muy útil, par uno de

nuestros objetivos principales que es la creación de una interfaz intuitiva para el uso del usuario de nuestro sistema.

El planificador de tareas base se encuentra en un fichero Python estándar debido a que no se pueden importar clases de ficheros I-Python Book. Esta es la mayor desventaja de éste sistema, aunque en la interfaz gráfica tienes una opción de descargar el I-Python Book como fichero estándar Python.

Capitulo 6: Arquitectura de la solución

En este apartado vamos a realizar la descripción de la solución del problema, que en este caso es la ejecución de tareas en una Granja de Ordenadores. Para ello vamos a abordar dos aspectos muy importantes.

En primer lugar, hablaremos de la comunicación de nuestro programa con la granja de ordenadores de la que disponemos.

En segundo lugar, nos dispondremos a revelar como está diseñado el planificador de tareas que realizará las gestiones pertinentes para poder enviar nuestras tareas a la granja de HPC.

6.1. Comunicación con la granja de HPC

Para realizar la comunicación con nuestro sistema de ordenadores hay varios pasos imprescindibles a completar para que la conexión y ejecutar los diversos ficheros que queremos. Los pasos están divididos en 3. La conexión, la petición y asignación de recursos y la ejecución de nuestro programa.

Ilustración 10 Diagrama de comunicaciones

6.1.1. Conexión

Para realizar la conexión, en primer lugar, nos conectamos al Zookeeper para averiguar cuál es el nodo maestro o máster activo de nuestro sistema distribuido. Esta comunicación está representada en el diagrama posterior mediante la flecha azul que une el planificador con el Zookeeper. En esta se puede apreciar como el planificador envía un mensaje de pregunta al Zookeeper.

Cuando recibe la pregunta, el Zookeeper, a su vez, vuelve a realizar esta misma a todos los nodos conectados a él. El nodo maestro activo responde con un mensaje al Zookeeper, identificándose como tal y le devuelve su dirección IP, en nuestro caso. El Zookeeper envía la información al planificador que realizó la petición y cierra la conexión con él.

Por último, el planificador se autentica en el máster, como usuario para finalizar el estado de conexión y poder realizar la petición de recursos. Este paso se realiza en la función *registered* de nuestra librería Python (dentro de la clase planificador).

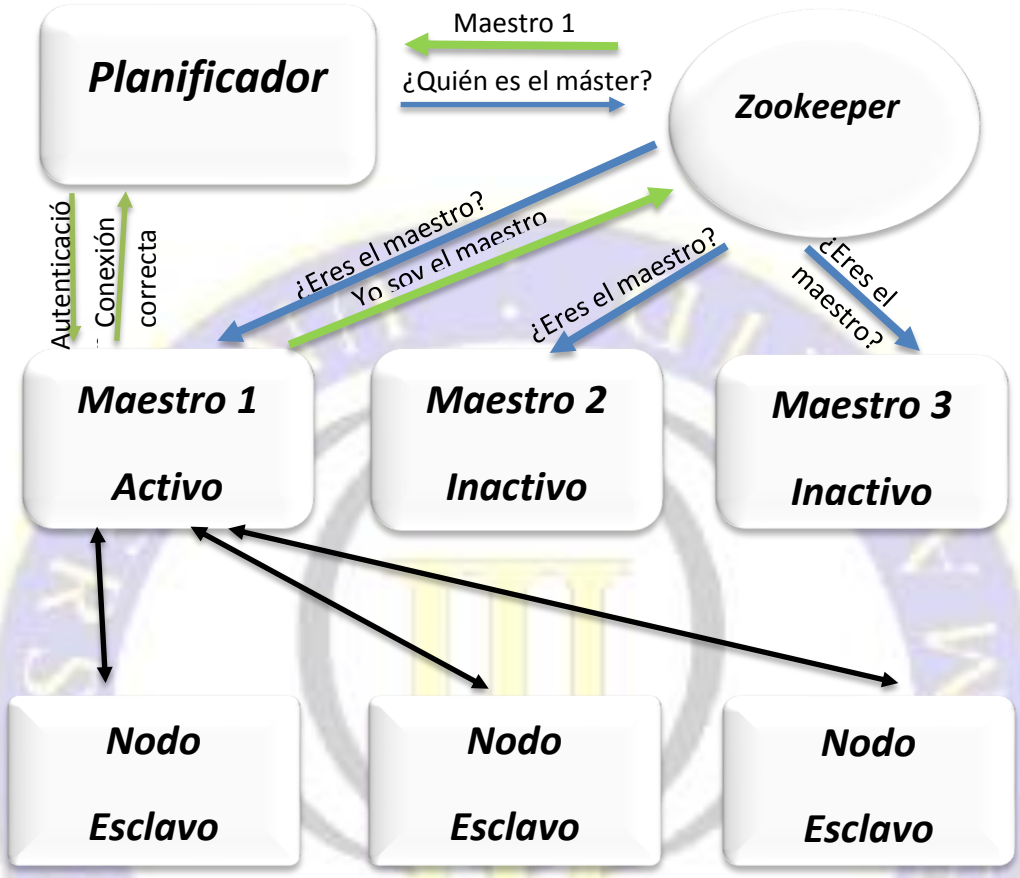

Ilustración 11 Conexiones

6.1.2. Petición y Asignación de recursos

Una vez que se ha realizado exitosamente la conexión con el máster correspondiente, el planificador le hace una petición de recursos. El nodo maestro les pide a todos sus nodos agentes, o esclavos, que le manden los recursos disponibles y éste a su vez se los manda al planificador.

En el interior del planificador, en la función *resourceOffers* se almacenan los datos recibidos y se elige el primer nodo esclavo que cumple los requisitos tanto de memoria como de CPUs exigido por cada tarea y se lo asigna a una de ellas aleatoriamente. Cuando ha elegido un nodo agente, el planificador le

manda la tarea a ejecutar al nodo maestro. A su vez, éste le manda la tarea al nodo esclavo elegido.

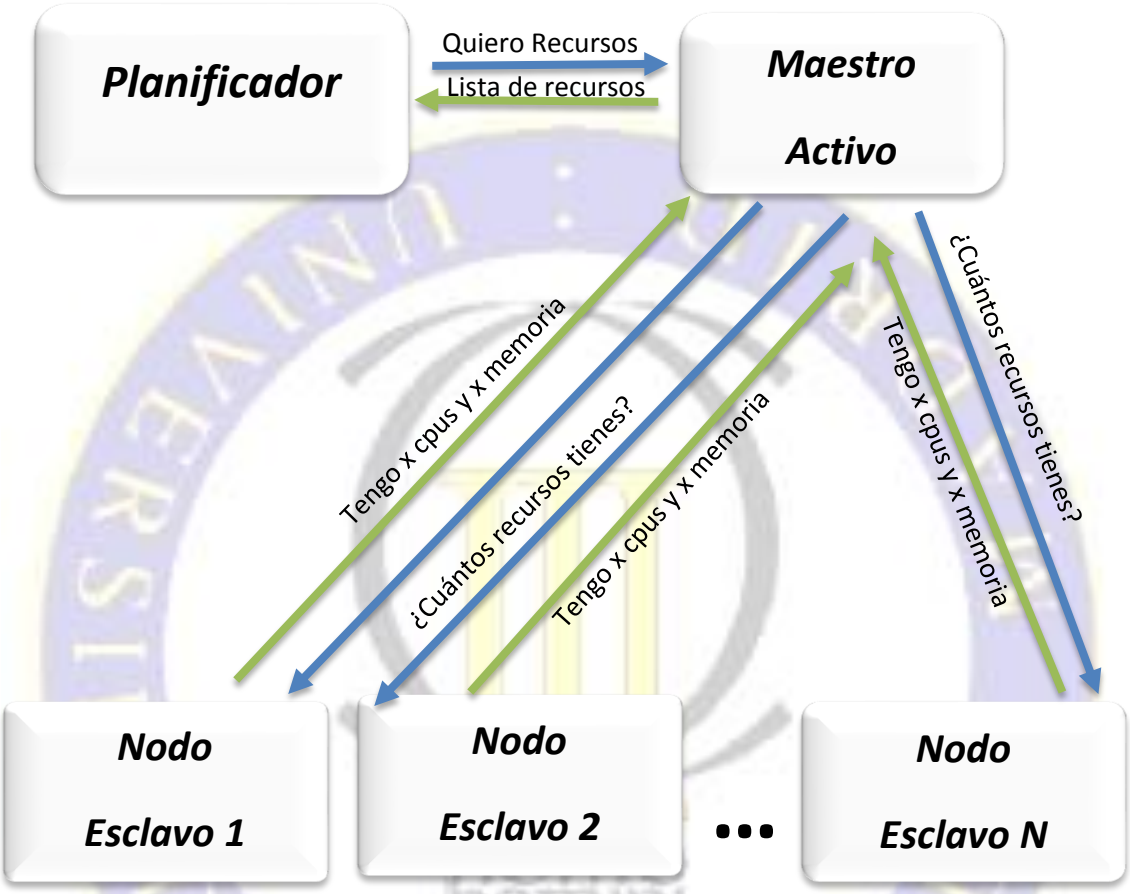

Ilustración 12 Petición y asignación de recursos

6.1.3. Ejecución

Una vez que los recursos han sido asignados, cada nodo esclavo ejecuta su tarea correspondiente, con el ejecutor, y devuelve al nodo maestro si ha finalizado correctamente o si ha habido un error y que tipo de error ha sido este. En nuestro planificador, tenemos programado que los estados de la ejecución se guarden en un fichero de salida, para visualizar más cómodamente los posibles errores en la simulación y ejecución de cada una de las tareas. Este control de estado se hace en *statusUpdate*. Así mismo, se recomienda al usuario, que realice un guardado de las soluciones de su fichero,

en otro fichero, dado que, nuestro programa no toma en consideración los resultados de los programas internos a las tareas. Es decir, la simulación es independiente del contenido de la tarea.

6.2. Planificador de tareas

En este apartado vamos a explicar las funciones que realiza un planificador de tareas. También ilustraremos como se realiza la comunicación con el ejecutor. Además de cada uno de los pasos que suceden para que una tarea se ejecute correctamente en nuestra granja de HPC.

6.2.1. Funciones principales

Las principales funciones de nuestro planificador de tareas son las siguientes.

En primer lugar, recibe los recursos con los que el usuario quiere que ejecutes su tarea. Cuando el planificador obtiene los recursos necesarios de un nodo agente o esclavo, por el mecanismo descrito en el apartado anterior [6.1.1, 6.1.2], crea una tarea. Para ello rellena los campos que identificamos en el apartado de **tarea [5.2.2]**, con los datos proporcionados usuario.

Una vez ha creado la tarea, se la manda al ejecutor que se describe más detalladamente en el apartado **[6.2.2].** Este es un software creado para la ejecución de tareas en los nodos agentes. Este mecanismo realizará las funciones descritas en el apartado **[6.1.3].**

Mientras la tarea se ejecuta, el planificador realiza el mismo proceso para el resto de tareas que posee el proceso en ejecución.

Cuando las tareas han finalizado el planificador anula la reserva de recursos y cierra la conexión con la granja de HPC devolviendo el estado de las tareas en un fichero.txt que ha sido introducido por el usuario.

6.2.2. Ejecutor

Este mecanismo **realiza la ejecución de cada tarea** en los nodos agentes y el **seguimiento del estado** de ésta. Cada nodo posee un ejecutor idéntico. El ejecutor se comunica con el planificador para devolver el estado de la tarea que se encuentra en su nodo y los diversos mensajes de error, y en el caso de que la tarea haya finalizado, si la finalización ha sido correcta. En el diagrama posterior veremos un diagrama de la arquitectura interna de Apache Mesos para el paso de mensajes cuando la conexión ha sido establecida y los recursos asignados.

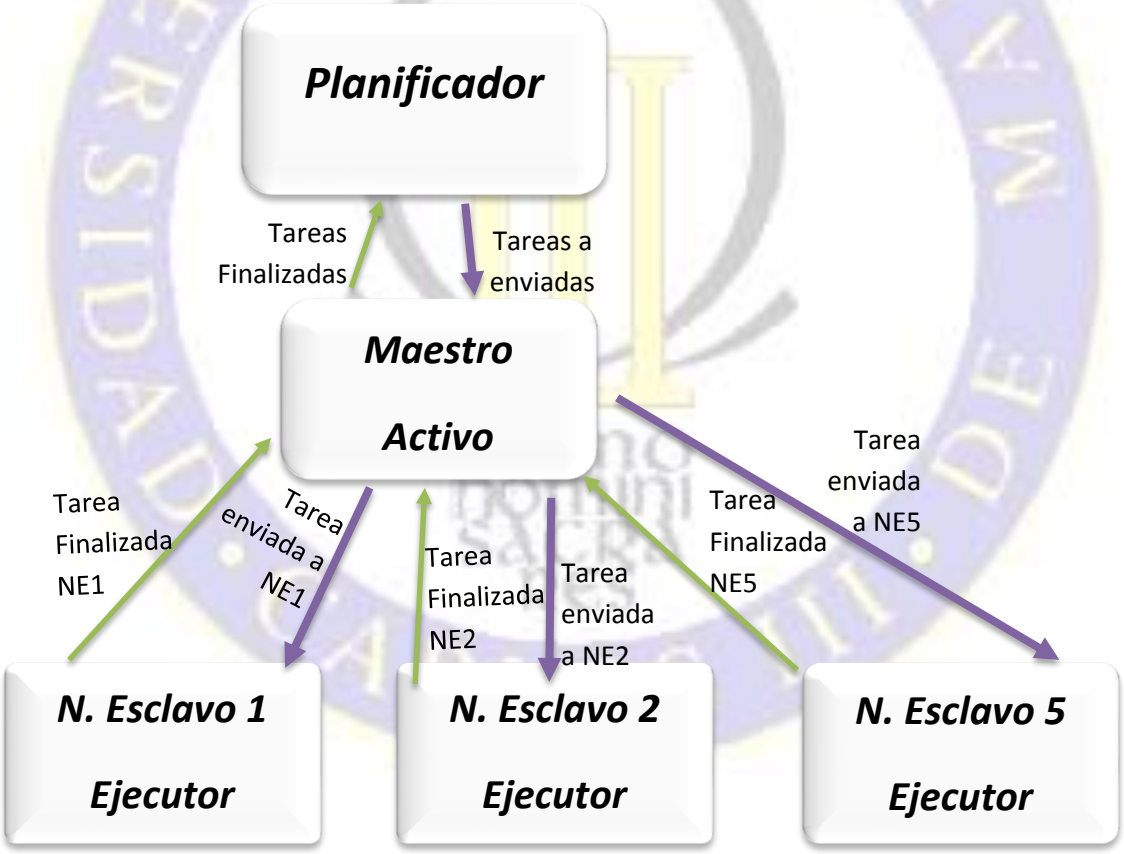

Ilustración 13 Ejecución de Tareas

Aunque hay posibilidad de usar un ejecutor propio en un futuro, para poder realizar un control mayor de los procesos y devolver mensajes de error internos de la tarea. En este proyecto vamos a utilizar el ejecutor básico que nos ofrece la plataforma.

La utilización del ejecutor propio conlleva esas ventajas, pero para poder usarlo se debe de instalar Python en los nodos esclavos. Esta desventaja produce un gasto de recursos en todos los nodos esclavos, y dado que, las ventajas no son tan amplias, se optó por usar el ejecutor por defecto existente en la instalación de Mesos. En un futuro, podría considerarse una mejora para el trabajo, la creación de un ejecutor competente para el procesado de tareas muy complejas, donde si sería muy útil.

Capitulo 7: Pruebas y evaluación de resultados

En este apartado del documento vamos a realizar una evaluación de los resultados obtenidos al ejecutar nuestro proyecto y las diversas pruebas a las que ha sido sometido.

7.1. Conexión con el sistema

En este apartado explicaremos diversas pruebas realizadas a través de la interfaz de usuario para comprobar que la conexión explicada en el apartado **[6.1.1]** es correcta.

En primer lugar, se realiza una conexión simple al nodo Zookeeper para que éste nos asigne el nodo maestro activo. De esta manera, aunque cambie el nodo principal obtendremos resultados favorables en nuestra simulación. Este procedimiento ha consistido en la simulación de una tarea con la dirección del Zookeeper en el bloque del proceso. La simulación de la tarea finalizó correctamente por lo que nuestra conexión se realiza de manera correcta.

Para determinar qué sucedería en caso de fallo, se realizó una simulación introduciendo una dirección de nodo maestro errónea. Esto nos llevó a una finalización de la conexión devolviendo como error en nuestro fichero de resultados, Master Unreachable.

7.2. Ejecución de tareas

Para comprobar si la **ejecución de tareas [6.1.3]** se realizaba correctamente, se realizaron ejecuciones de los dos primeros ficheros de prueba adjuntos en el apartado de Anexos.

En primer lugar, se realizaron pruebas realizando la misma tarea en varias CPUs comprobando que la creación de las tareas se realiza correctamente y que cada una de ellas está asignada a un nodo. En este caso se ha visualizado que los estados de las tareas, tenían una finalización correcta, a través de los mensajes que envía el **ejecutor [6.2.2]** a nuestro planificador, y que los resultados de los estados habían sido recogidos correctamente en el fichero de resultados. Pudiendo observar en él los diferentes estados por los que ha pasado una tarea.

En segundo lugar, hemos realizado simulaciones con tareas de diferente tamaño para comprobar que sucede cuando terminan las tareas y deben esperar por otra. Con las simulaciones realizadas observamos en el fichero de salida que cuando se finalizan las tareas de menor peso de ejecución se guarda su estado y espera a que finalicen las tareas con mayor peso en la ejecución para finalizar el proceso y cerrar la conexión.

También debemos añadir en este apartado, el funcionamiento de nuestro programa cuando una tarea finaliza de manera incorrecta. Hemos decidido por cuestiones técnicas de facilidad de implementación que el resto de tareas finalice abruptamente y se repita la ejecución del proceso en cuestión.

7.3. Asignación de recursos y verificación

Para saber si la **asignación de recursos [6.1.2]** se realiza de manera correcta a las tareas se han hecho varias simulaciones cambiando el número de recursos asignados a las tareas.

En primer lugar, realizamos una simulación de dos procesos independientes utilizando un mayor número de CPUs en el segundo proceso. Para ello en nuestro fichero interfaz de usuario incluido en el anexo Batería de pruebas, hemos modificado las variables CPUs y CPUs2 que contienen el número de CPUs necesarias para ejecutar las tareas del proceso 1 y 2, respectivamente. Con este sencillo cambio respecto a las simulaciones normales, podemos observar en los tiempos de ejecución que aparecen en el fichero como la velocidad cambia proporcionalmente inversa, es decir si un proceso posee el doble de CPUs la velocidad será la mitad, con respecto al otro proceso.

En el caso del ejemplo del anexo podemos ver que al aumentar recursos de memoria el tiempo de ejecución se divide. Para realizar este cambio hemos modificado las variables MEMORY y MEMORY2 en el apartado de variables necesarias para ejecutar tareas. Estas variables se corresponden con los recursos de memoria que se asignan en cada CPU. En este caso no es totalmente proporcional debido a la poca complejidad de los procesos, que nos lleva a tiempos de procesamiento infinitesimales.

En los casos en los que no existen recursos disponibles, el proceso sigue activo hasta que se liberen los recursos necesarios para ejecutar las tareas. Esto nos puede llevar a casos de bloqueos si los recursos del sistema están ocupados. Dentro de la librería existe la opción de finalizar el proceso, pero no se ha implementado por la posibilidad de que sea un bloqueo pasajero. Si el bloque o perdura siempre se puede finalizar el proceso de manera abrupta finalizando la ejecución de nuestro I-Python Book que nos sirve de interfaz.

Una vez han sido determinados todos los posibles errores del sistema y las pruebas y mejoras realizadas para que nuestra librería sea lo más eficientemente posible vamos a enumerar las conclusiones del documento y los posibles trabajos futuros que serían interesantes contemplar.

Capitulo 8: Conclusions and Future works

8.1. Fundamental conclusions

As for the completion of our project, we can conclude that both the library to create, as the user interface were successful. This is in part occurred for the decisions explained before in this document.

The use API Apache Mesos helped with issues of connection with our distributed system and to make the better distribution of executable elements in different sub-processes or tasks.

They have met the requirements required compatibility with existing systems in the system that we have and minimization of changes in structure.

8.2. Future works

Future improvements that can be considered are different due to the simplicity of the system.

One of them, is the use of an own executor of tasks, in this work is used the generated by default in Apache Mesos. This would give us more control in the tasks states within the cluster.

Another improvement that can be implemented in the future is a queue manager to determine the priorities of connecting multiple users in the cluster. Today is not necessary because the existing actual blocking rate is very low, and the complexity of the problem is too broad.

Capitulo 9: Conclusiones y trabajos futuros

9.1. Conclusiones principales

En cuanto a la finalización de nuestro proyecto podemos concluir que tanto la biblioteca a crear, como la interfaz de usuario se realizaron correctamente aprovechando las ventajas de las elecciones que realizamos en el trabajo, como pueden ser las herramientas utilizadas y el tipo de gestor de procesos.

La utilización de la API de Apache Mesos nos ayudó con las cuestiones de conexión con nuestro sistema distribuido y a realizar la distribución de los elementos ejecutables en diferentes sub-procesos o tareas.

Se han cumplido los requisitos exigidos de compatibilidad con los sistemas existentes en el sistema del que disponemos y de minimización de cambios en la estructura.

9.2. Futuros trabajos

Las futuras mejoras que se pueden considerar son varias debido a la simplicidad del sistema.

Una de ellas consiste en la utilización de un ejecutor de tareas propio, en el trabajo se utiliza el generado por defecto en Apache Mesos. Esto nos daría más control de las tareas en sus estados dentro del sistema.

Otra mejora que se puede implementar en un futuro es un gestor de colas que determine las prioridades de conexión de los múltiples usuarios de la granja de HPC. En la actualidad no es necesario dado que con el existente la tasa de bloqueo real es muy baja, y la complejidad del problema es demasiado amplia.

Capitulo 10: Memoria económica

En este apartado vamos a realizar el modelo económico de nuestro proyecto en el que determinaremos los diferentes costes de nuestro proyecto y la planificación del tiempo empleado para realizarlo.

*10.1.*Contexto Socio-económico

En el contexto socio económico cabe destacar un entorno de crisis que fomenta el aprovechamiento de los recursos disponibles y nos ocasiona dificultades para obtener financiación para proyectos innovadores y con la utilización de nuevas herramientas

También hay que destacar el marco social en el que nos encontramos, donde la educación es un pilar importante para la sociedad y el reconocimiento general. Este es un factor a nuestro favor, pero ante un contexto económico negativo, no es lo suficientemente importante para favorecernos.

*10.2.*Presupuesto del proyecto

Para la realización de este apartado hemos tenido varios factores en consideración. Uno de ellos ha sido la implicación humana, el otro de ellos es el coste energético de los recursos usados para realizar este proyecto. No se ha contemplado el coste de fabricación del sistema debido a que era un sistema ya existente anterior a la creación del proyecto.

Presupuesto				
	Recursos	Precio(€/h)	Tiempo (h)	Total
luz	100,00 kW	0,11 €	480	5.280,00 €
personal	Ingeniero 1	10,00 €	480	4.800,00 €
	ingeniero 2	8,00 €	480	3.840,00 €
Subtotal				13.920,00 €
Riesgos				696,00 €
Presupesto antes de impuestos				14.616,00 €
IVA				3.069,36 €
Presupuesto final				17.685,36 €
El precio de luz que figura es el precio medio de KW/h del 23/9/2016				

10.3. Planificación del tiempo

Dado que la realización en el tiempo ha sido de manera discontinua en este apartado realizaremos la representación de la planificación mediante un diagrama de Pert que representa el tiempo asignado a cada tarea. Para realizar este tipo de planificación previamente necesitamos una tabla que nos ordene los diferentes

Nombre clave	Tipo	Duración	T. inicio Early	T.fin Early	Actividad Anterior	T inicio Last	T fin Last
A	Comienzo del proyecto	1	0	1	-	0	1
B	Instalación de las herramientas	5	1	6	A	1	6
C	Comprobación de funcionamiento	2	6	8	B	6	8
D	Aprendizaje del lenguaje	7	1	8	A	1	11
E	Familiarización con la arquitectura	3	8	11	C	8	11
F	Creación de la librería	20	11	31	D, E	11	31
G	Creación de la interfaz de usuario	15	11	26	D, E	11	31
H	Búsqueda de información Memoria	10	31	41	F, G	31	41
I	Realización de Pruebas	6	31	37	F, G	31	53
J	Redacción de la memoria	12	41	53	H	41	53
K	Realización de la presentación	2	53	55	J, I	53	55
L	Preparación de la presentación	5	55	60	K	55	60
* La representación de los números se corresponde a días empleados en esos proyectos							

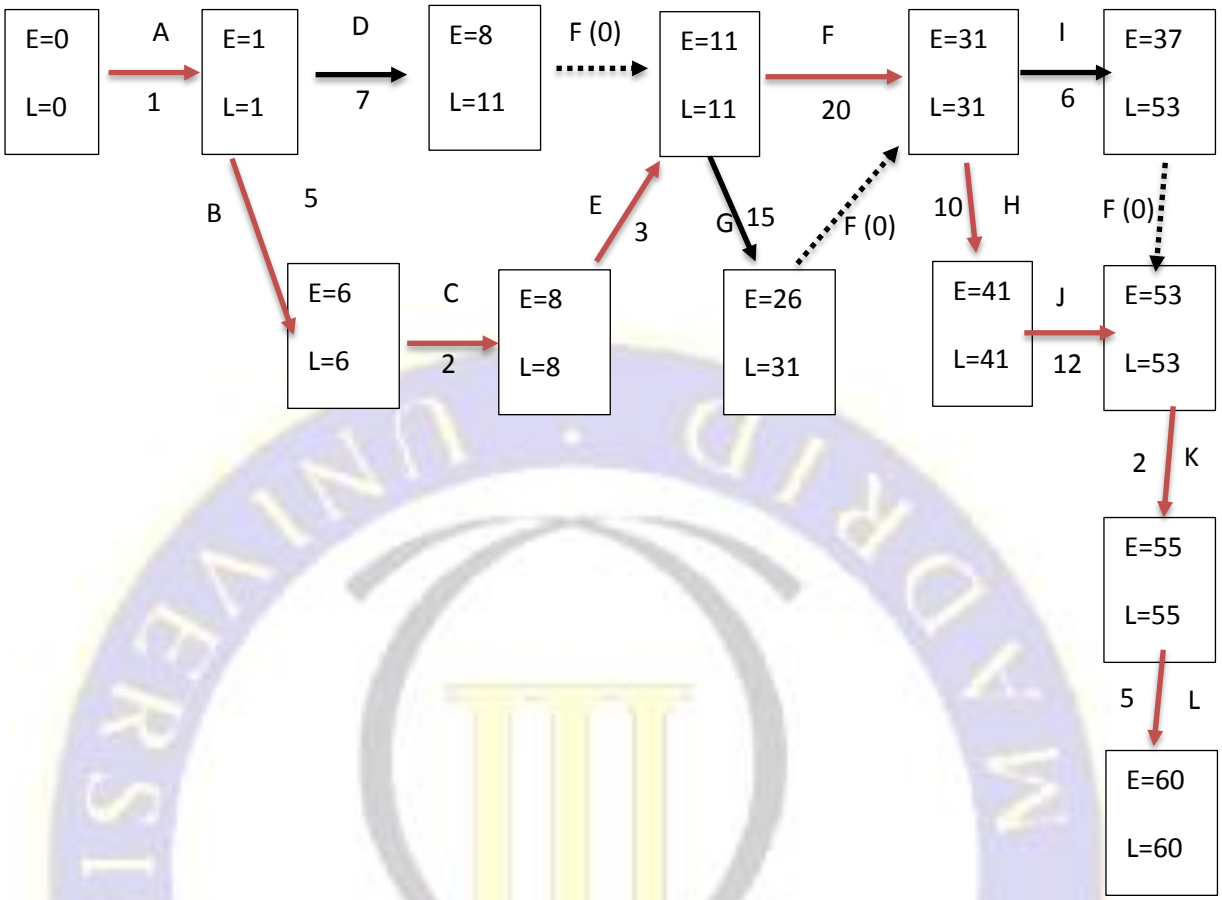

Ilustración 14 Diagrama de Pert del proyecto

Como podemos observar el tiempo estimado de duración del proyecto son 60 días. Este proceso se alterará si alguna tarea del camino crítico sufre algún retraso. Este camino es el que contiene las flechas del diagrama en rojo. Esto sucede porque las tareas son dependientes unas de otras.

Para determinar el tiempo que podemos tener para realizar nuestras tareas sin que afecte a el tiempo de finalización vamos a calculas los tres tipos de holguras de los que disponemos.

La holgura total es el tiempo que puede tardar más la tarea si se inicia en el tiempo previsto, para que finalice sin realizar ningún retraso en el proyecto.

La holgura libre es la que disponemos cuando un proyecto empieza y acaba en los tiempos estimados.

Y por último la holgura independiente de las tareas es aquel caso en el que las tareas empiezan en su tiempo más lejano (T. inicial Last) y finalizan en el tiempo más cercano.

Como podemos ver en la figura posterior en nuestro proyecto las holguras libres e independientes son iguales a 0 en todas las tareas, en cambio en la tarea D, G e I existe una holgura total no nula debido a que la siguiente tarea que depende de ellas depende también de otras anteriormente que finalizan después de éstas.

Actividades	Holgura total	Holgura libre	Holgura independiente
A	0	0	0
B	0	0	0
C	0	0	0
D	3	0	0
E	0	0	0
F	0	0	0
G	5	0	0
H	0	0	0
I	16	0	0
J	0	0	0
K	0	0	0
L	0	0	0

Capitulo 11: Referencias y Bibliografía

*11.1.*Bibliografía de la Memoria:

11.1.1. G. Colouris, J. Dollimore, T. Kindberg. Addison Wesley **Sistemas distribuidos: Conceptos y diseño**. (2001) ISBN 84-7829-049-4

11.1.2. A. Tanenbaum**. Sistemas operativos distribuidos.** Prentice Hall (1996). ISBN: 968-880-627-7

11.1.3. Cap: 1 1. Introducción a las Arquitecturas de Altas Prestaciones. *Arquitecturas de Altas Prestaciones***:** Fecha de consulta (3-08-2016). Disponible en: http://geneura.ugr.es/~jmerelo/asignaturas/AAP/AAP-Tema-1.mhtml

11.1.4. **Introducción a la computación distribuida**. Dto. De Matemáticas y Computación. Grado en Ingeniería informática. Asig. Sistemas Distribuidos. Fecha de consulta (3-08-2016) **[En línea]**Disponible en: http://www.unirioja.es/cu/fgarcia/sd/pub/teo/01-IntroduccionALaComputacionDistribuida.pdf

11.1.5. **Art. Concepto de Computación.** Aut. Anónimo. Fecha de consulta: 3/09/2016 Disponible en: http://concepto.de/computacion/#ixzz4JC8pfGMA

11.1.6. Anónimo **Definición de informática - Qué es, Significado y Concepto** Aut. Fecha de consulta: 3/09/2016 Disponible en: http://definicion.de/informatica/#ixzz4JCAteKSS

11.1.7. **Documentation Mesos**. Fecha de última consulta: 10/9/2016 Disponible en: http://mesos.apache.org/documentation/latest/

11.1.8. E. Estrella. Supercomputadoras. Fecha de consulta: 6/8/2016 **[En línea]**Disponible en: http://www.monografias.com/trabajos65/supercomputadoras/supercomputadoras.shtml

11.1.9. Artículo de la Wikipedia: Granja de HPC (High performance computation). **[En línea]** Disponible en: https://es.wikipedia.org/wiki/Cl%C3%BAster_(inform%C3%A1tica)

11.1.10. Anónimo. **What is HTCondor?** Fecha de consulta 10/9/2016 **[En línea]** Disponible en: http://research.cs.wisc.edu/htcondor/description.html

11.1.11. José A. Gómez G. **Computación Cooperativa**. Universidad Católica \Nuestra Señora de la Asunción. Facultad de Ciencias y Tecnologías. Departamento de Electrónica e Informática. Fecha de presentación: 13 de octubre de 2015. Fecha de consulta 10/9 /2016 **[En línea]** Disponible en: http://jeuazarru.com/wp-content/uploads/2015/11/Computacion-Cooperativa.pdf

11.1.12. Anónimo. **Grid computing** Disponible en: https://en.wikipedia.org/wiki/Grid_computing

11.1.13. Anónimo. **Sun Grid Engine** (Artículo de la Wikipedia) Fecha de consulta: 10/8/2016 Disponible en: https://es.wikipedia.org/wiki/Sun_Grid_Engine

11.1.14. **Mesos: A Platform for Fine-Grained Resource Sharing in the Data CenterMesos** Autores: Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion

Stoica (University of California, Berkeley) Thursday 30th September, 2010, 12:57

*11.2.*Bibliografía del código:

11.2.1. Capítulo 10. Un paseo por los módulos de la librería estándar "Python para principiantes" **[En Línea]** Disponible en: http://librosweb.es/libro/python/capitulo_10/modulos_de_sistema.html

11.2.2. **Documentation Mesos**. Fecha de última consulta: 10/9/2016 Disponible en: http://mesos.apache.org/documentation/latest/

*11.3.*Bibliografia imagenes

11.3.1. **Top 500** fecha de consulta 8/9/2016 Disponible en: https://www.top500.org

11.3.2. Chinese supercomputer is the world's fastest — and without using US chips http://www.theverge.com/2016/6/20/11975356/chinese-supercomputer-worlds-fastes-taihulight

11.3.3. Entrada de blog sistemas distribuidos Disponible en: https://gmcastillob.wordpress.com/2013/02/17/sistemas-distribuidos/

11.3.4. Render de servers de Apple Disponible en: http://www.soydemac.com/la-gran-granja-de-servidores-de-apple/

11.3.5. Computación paralela Disponible en: https://es.wikipedia.org/wiki/Computaci%C3%B3n_paralela

11.3.6. Tianhe-2, Most Powerful Supercomputer in the World, Runs Ubuntu. Disponible en: http://news.softpedia.com/news/tianhe-2-most-powerful-supercomputer-in-the-world-runs-ubuntu-487271.shtml#ixzz4KEv4cDAa

11.3.7. Esquema de Computación distribuida [En línea] Disponible en: https://mind42.com/public/448bd6ef-d880-4da9-b3de-78223f469798

Capitulo 12: Palabras clave

12.1. **Script:** Fichero de datos que contiene código en un lenguaje de programación compilable.

12.2. **Granja de HPC (High performance computation) :** Red de equipos interconectados para la simulación de diferentes tareas de diversos usuarios.

12.3. **Fairness:** Injusticia, los problemas de equidad de utilización de recursos en los diferentes sistemas de acceso se denominan con este nombre.

12.4. **Hilos (threads):** Los hilos o threads (en inglés) son diferentes procesos de ejecución en los que se dividen las diferentes sub-tareas

12.5. **CPU:** Unidad de procesamiento automático de información.

12.6. **Scheduler o Planificador de tareas:** Herramienta que realiza la gestión de procesos en un sistema distribuido.

12.7. **Middelware:** software en el que se basa la computación en rejilla para garantizar la transparencia entre los diferentes dispositivos de la red de computación cooperativa.

12.8. **Procesos mono-hilio y multi-hilo:** los procesos mono-hilo son aquellos que solo disponen una tarea de ejecución y los procesos multi-hilo son aquellos que se componen de multiples subtareas que se ejecutan mediante hilos de manera paralela

12.9. **Entorno multiusuario:** entorno en el que muchas personas participan en el uso de recursos compartidos.

Capitulo 13: Anexos

En este apartado encontraremos el código del proyecto implementado en Python. Y explicado en el interior del documento.

*13.1.*Planificador de tareas

En este anexo encontraremos el código perteneciente al planificador de tareas. En él se pueden encontrar funciones del gestor de colas, debido a que es un gestor simple y era más práctico unir ambas funcionalidades en un único código.

```
# coding: utf-8

"""Este código contiene un planificador de tareas básico
para la ejecución en el entorno del clúster del departamento
de Teoria de la Señal de la universidad Carlos III de Madrid"""

"""A continuación, se realiza las
importaciones de las diversas librerias
de python para el correcto funcionamiento
del programa"""

import os
import sys
import time

import mesos.interface
from mesos.interface import mesos_pb2
import mesos.native

""" Para la realización de este proyecto usamos varios objetos predeterminados en mesos.
El objeto Framework se identifica con el proceso, el objeto task con la tarea, el objeto operación con el programa en ejecución, y el objeto resources con los recursos disponibles """

""" En este apartado se encuentra el código que implementa
```

```
el planificador de tareas de la granja de ordenadores.
Posee 4 módulos en los que se inicializa el planificador de tareas,
se registra el proceso, se le asignan recursos a cada tarea
(tales como las CPUs que ejecutan cada tarea y la memoria de cada una de ellas),
y se recupera el estado de el proceso hasta la finalización de éste en un fichero aparte """

class Planificador(mesos.interface.Scheduler):

    """ Inicia las principales variables del planificador de tareas """
    def __init__(self,fich,T_totales,CPUs,MEMORY):


        self.taskData = {}
        self.tasksLaunched = 0
        self.tasksFinished = 0
        self.T_totales=T_totales
        self.CPUs=CPUs
        self.MEMORY=MEMORY
        self.taskrunning=0
        self.fich=fich

        update = 0
    #Este apartado se encarga de realizar el registro en el nodo maestro
    def registered(self, driver, frameworkId, masterInfo):
        print "Registered with framework ID %s" % frameworkId.value

    #en este módulo hacemos la asignación de recursos a las tareas del proceso
    def resourceOffers(self, driver, offers):
        for offer in offers:
            tasks = []
            offerCpus = 0
            offerMem = 0
            #en el siguiente bucle se guardan los recursos que te devuelve el nodo maestro
            for resource in offer.resources:
                if resource.name == "cpus":
                    offerCpus += resource.scalar.value
                elif resource.name == "mem":
                    offerMem += resource.scalar.value



            remainingCpus = offerCpus
            remainingMem = offerMem
```

```
        #realiza la asignación del nodo esclavo a la tarea y la crea con los datos necesarios
        while self.tasksLaunched < self.T_totales and remainingCpus >= self.CPUs and remainingMem >= self.MEMORY:
            tid = self.tasksLaunched
            self.tasksLaunched += 1

            print "Launching task %d using offer %s"  % (tid, offer.id.value)

            task = mesos_pb2.TaskInfo()
            task.task_id.value = str(tid)
            task.slave_id.value = offer.slave_id.value
            task.name = "task %d" % tid

            cpus = task.resources.add()
            cpus.name = "cpus"
            cpus.type = mesos_pb2.Value.SCALAR
            cpus.scalar.value = self.CPUs

            mem = task.resources.add()
            mem.name = "mem"
            mem.type = mesos_pb2.Value.SCALAR
            mem.scalar.value = self.MEMORY

            #EJECUTA EL CÓDIGO QUE SE ENCUENTRA EN EL FICHERO EN CADA TAREA

            task.command.value =self.fich[self.tasksLaunched-1]
            tasks.append(task)


            remainingCpus -= self.CPUs
            remainingMem -= self.MEMORY

        #Estos comandos guardan las variables necesarias para mandar
        #la tarea a la granja de ordenadores
        operation = mesos_pb2.Offer.Operation()
        operation.type = mesos_pb2.Offer.Operation.LAUNCH
        operation.launch.task_infos.extend(tasks)

        #envia al nodo maestro el nodo reservado y la tarea a ejecutar
        driver.acceptOffers([offer.id], [operation])
```

```
#Modulo de seguimiento de la tarea
def statusUpdate(self, driver, update):

    print "Task %s is in state %s" % (update.task_id.value, mesos_pb2.TaskState.Name(update.state))

    #comprueba si la tarea ha finalizado y en caso de que todas
    #lo hayan hecho finaliza el programa
    if update.state == mesos_pb2.TASK_FINISHED:

        self.tasksFinished += 1

        if self.tasksFinished == self.T_totales:
            print "All tasks done"

            driver.stop()

    #comprueba si hay una tarea erronea y si la hay finaliza el programa
    if update.state == mesos_pb2.TASK_LOST or update.state == mesos_pb2.TASK_KILLED or update.state == mesos_pb2.TASK_FAILED:
        print "Aborting because task %s is in unexpected state %s" % (update.task_id.value, mesos_pb2.TaskState.Name(update.state))
        driver.abort()
```

13.2. Batería de pruebas

Este fichero contiene las diversas pruebas explicadas en el Apartado de **pruebas y gestión de resultados [Capitulo 7:].**

La primera prueba que se ha realizado consiste en la simulación de dos procesos asignándole el doble de memoria en cada cpu a sus tareas internas.

```
# coding: utf-8

import os
import sys
import time

import mesos.interface
from mesos.interface import mesos_pb2
import mesos.native
from PlanificadorDeTareas import Planificador
```

```
""" Variables a definir """
salida ='salida.txt' #fichero donde quieres que te devuelva los estados de ejecución de las tareas

"""En este apartado se encuentran las variables globales del archivo:
   El número de tareas a lanzar en el cluster= T_totales
   El número de equipos a usar por cada tarea= CPUs
   La cantidad de memoria que va a usar cada tarea en la cpu correspondiente=MEMORY
   """
T_totales = 5          #numero de tareas en el proceso 1
CPUs =10               #numero de CPUs usadas para cada tarea en el proceso 1
MEMORY = 600           #numero de Memoria usada para cada tarea en el proceso 1
T_totales2 = 5                 #numero de tareas en el proceso 2
CPUs2 =10              #numero de CPUs usadas para cada tarea en el proceso 2
MEMORY2 = 1200 #numero de Memoria usada para cada tarea en el proceso 2
fich =[]               #variable en la que se guardará el fichero a ejecutar en cada tarea proceso 1
fich2 =[]        #variable en la que se guardará el fichero a ejecutar en cada tarea proceso 2

#bucle para guardar el mismo fichero en cada tarea.
for i in range(T_totales):
   fich.append('exec /export/usuarios01/eexposito/Datos/bin/prueba.sh')
for i2 in range(T_totales2):
   fich2.append('exec /export/usuarios01/eexposito/Datos/bin/prueba.sh')


if __name__ == "__main__":

   #guarda los datos en un fichero para poder
   #ver los errores y estados de finalización de las tareas
   sys.stdout=open(salida,'w')

   #crea los procesos
   framework = mesos_pb2.FrameworkInfo()
   framework.user = ""              #Ejecuta como usuario en le que está el fichero
   framework.name = "Framework"        #nombre del proceso

   framework.principal = "test-framework-python"

   tiempo_inicial1 = time.time()    #tiempo en el que se inicia el porceso1
```

```
    #Abajo se puede ver el ejecutor del planificador al que le tienes que
    #pasar por parametros el planificador usado, el proceso a ejecutar
    #y la dirección del Zookeeper
    driver = mesos.native.MesosSchedulerDriver(
        Planificador(fich,T_totales,CPUs,MEMORY),    #libreria    creada    por
nosotros basada API Mesos
        framework,
        "zk://10.0.12.58:2181,10.0.12.59:2181/mesos")
    tiempo_final1 = time.time()      #tiempo final de ejecución de las tareas

    tiempo_ejecucion1 = tiempo_final1 - tiempo_inicial1 #tiempo trancurrido para
la conexión
    #comprueba el cierre de la conexion con nuestro sistema distribuido
    status = 0 if driver.run() == mesos_pb2.DRIVER_STOPPED else 1
    #imprime el tiempo de ejecucion proceso 1
    print 'El tiempo de ejecucion fue:',tiempo_ejecucion1       #En segundos

    tiempo_inicial2 = time.time()    #tiempo en el que se inicia el porceso1

    #Abajo se puede ver el ejecutor del planificador al que le tienes que
    #pasar por parametros el planificador usado, el proceso a ejecutar
    #y la dirección del Zookeeper

    driver2 = mesos.native.MesosSchedulerDriver(
        Planificador(fich2,T_totales2,CPUs2,MEMORY2),
        framework,
        "zk://10.0.12.58:2181,10.0.12.59:2181/mesos")

    tiempo_final2 = time.time()      #tiempo final de ejecución de las tareas

    tiempo_ejecucion2 = tiempo_final2 - tiempo_inicial2

    # comprueba si la conexión finalizó
    status += status if driver2.run() == mesos_pb2.DRIVER_STOPPED else 1


    #imprime el tiempo de ejecucion del proceso 2
    print 'El tiempo de ejecucion fue:',tiempo_ejecucion2 #En segundos
    #cierra la conexion
    driver.stop();

    sys.exit(status)
```

En la siguiente prueba se va a realizar la simulación de un fichero con mayor carga computacional para ver los diferentes retrasos cuando las tareas internas del proceso son diferentes y como, hasta que no terminan todas las tareas en ejecución del proceso éste no finaliza.

```
""" Variables a definir """
salida2 ='salida2.txt'          #fichero donde quieres que te devuelva los estados de ejecución de las tareas
fich =[]                        #variable en la que se guardará el fichero a ejecutar en cada tarea proceso
#fichero a ejecutar en tarea 1
fich.append('exec /export/usuarios01/eexposito/Datos/bin/prueba.sh')
#fichero a ejecutar en tarea 2
fich.append('exec /export/usuarios01/eexposito/Datos/run_test')
#fichero a ejecutar en tarea 3
fich.append('exec /export/usuarios01/eexposito/Datos/bin/prueba.sh')

"""En este apartado se encuentran las variables globales del archivo:
    El número de tareas a lanzar en el cluster= T_totales
    El número de equipos a usar por cada tarea= CPUs
    La cantidad de memoria que va a usar cada tarea en la cpu correspondiente=MEMORY
    """
T_totales = 3
CPUs = 10
MEMORY = 2000



if __name__ == "__main__":

    #guarda los datos en un fichero para poder
    #ver los errores y estados de finalización de las tareas
    sys.stdout=open(salida2,'w')

    #crea el proceso
    framework = mesos_pb2.FrameworkInfo()
    framework.user = " "            #Ejecuta como usuario en el que está el fichero
    framework.name = "Framework"        #nombre del proceso

    framework.principal = "test-framework-python"
    tiempo_inicial1 = time.time()    #tiempo en el que se inicia el porceso

     #Abajo se puede ver el ejecutor del planificador al que le tienes que
    #pasar por parametros el planificador usado, el proceso a ejecutar
    #y la dirección del Zookeeper

    driver = mesos.native.MesosSchedulerDriver(
```

```
    Planificador(fich,T_totales,CPUs,MEMORY),#libreria creada por nosotros basada API Mesos
    framework,
    "zk://10.0.12.58:2181,10.0.12.59:2181/mesos")

  tiempo_final1 = time.time()      #tiempo final de ejecución de las tareas

  #tiempo trancurrido para la conexión
  tiempo_ejecucion1 = tiempo_final1 - tiempo_inicial1

  # comprueba si la conexión finalizó
  status = 0 if driver.run() == mesos_pb2.DRIVER_STOPPED else 1

  #imprime el tiempo de ejecucion del proceso
  print 'El tiempo de ejecucion fue:',tiempo_ejecucion1 #En segundos
  #cierra la conexion
  driver.stop();

  sys.exit(status)
```

13.3. Ficheros de resultados

Prueba 1:

```
Registered with framework ID 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-0031
Launching task 0 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15022702
Launching task 1 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15022703
Launching task 2 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15022705
Launching task 3 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15022706
Launching task 4 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15022707
Task 2 is in state TASK_RUNNING
Task 4 is in state TASK_RUNNING
Task 0 is in state TASK_RUNNING
Task 3 is in state TASK_RUNNING
Task 1 is in state TASK_RUNNING
Task 2 is in state TASK_FINISHED
Task 4 is in state TASK_FINISHED
Task 0 is in state TASK_FINISHED
Task 3 is in state TASK_FINISHED
Task 1 is in state TASK_FINISHED
All tasks done
El tiempo de ejecucion fue: 0.00354695320129
Registered with framework ID 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-0032
Launching task 0 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15024557
Launching task 1 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15024559
Launching task 2 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15024560
Launching task 3 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15024561
Launching task 4 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O15024563
Task 2 is in state TASK_RUNNING
```

Task 0 is in state TASK_RUNNING
Task 1 is in state TASK_RUNNING
Task 3 is in state TASK_RUNNING
Task 4 is in state TASK_RUNNING
Task 2 is in state TASK_FINISHED
Task 0 is in state TASK_FINISHED
Task 1 is in state TASK_FINISHED
Task 4 is in state TASK_FINISHED
Task 3 is in state TASK_FINISHED
All tasks done
El tiempo de ejecucion fue: 0.000409126281738

Prueba 2

Registered with framework ID 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-0030
Launching task 0 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O14972979
Launching task 1 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O14972980
Launching task 2 using offer 10c70aef-f13c-4a4f-bdd3-3c2c1efbcd20-O14972981
Task 1 is in state TASK_RUNNING
Task 0 is in state TASK_RUNNING
Task 2 is in state TASK_RUNNING
Task 0 is in state TASK_FINISHED
Task 2 is in state TASK_FINISHED
Task 1 is in state TASK_FINISHED
All tasks done
El tiempo de ejecucion fue: 0.000877141952515

13.4. Extended abstract.

In this paper we have developed a library of process management and a user interface for a cluster. This project is created by the need to find a tool for simulating tasks in a multiuser environment.

13.4.1. Context

To determine what type of computation is the best for our problem we explain the four types of computations existing today.

The first is the monolithic computing. This is done using a single processing unit. To address the problem of multi-user performs a division of resources for time, assigning each user a certain time. Due to the delay problems generated by this system when the number of users is high, this has been first discarded choice.

The second option is the use of parallel computing. This system involves the use of multiple CPUs. The central computer divides each user task into smaller tasks so that the simulation at a higher speed, and divides of resources through time division. Assigning each sub-task to small amount time processing. The various processing threads communicate through networks, both internal and external shared memory (depending on the architecture), which perform the transfer of information to process multiple tasks through the mechanism described above. Still the multiuser problem continued existing. When the number of processing units is very large (millions of units) these teams are supercomputers, of which will talk more later.

The third option is the use of cooperative computing. This type of computer uses a mechanism called grid computing. This mechanism established communication between millions of computers, transparently in various parts of the world. Users belonging to the group simulation, give the remaining CPU resources to simulate large-scale projects. One of the projects being carried out today with this system is the search for extraterrestrial life. The main drawback of this system is the variability of available resources.

Finally, we will explain the chosen option for our system. Distributed computing is based on the use of multiple simple computers to perform our work. Each user is assigned a distributed system resources, which previously requested. This mechanism resource that generated the division in time, but we can carry blocking state. These cases occur when a user occupies the entire of resources through a process of great weight system. While the process is running, other users can´t access the system. Another problem is the Fairness generated, this problem is access to resources for user only. To solve these problems, exist queue management mechanisms and priorities. In our project, being a small environment, these mechanisms are not necessary.

Once you have explained the type of computing chosen we will explain our distributed system. Our system is a set of interconnected computers HPC. These teams are called HPC (high density computer). These computers are able to parallelize tasks inside, to reduce the processing time of these. This makes our system meets distributed computing and parallel computing to become more efficient.

Finally, we cannot make a full context of our project if we do not speak of its main competitor in the field of large-scale computing, supercomputers.

Supercomputers are teams with millions of CPUs that it have large amounts of memory to simulate various processes and perform their calculations using parallel computing. They are more powerful than computers farms in terms of available resources and memory. It also has no blocking problems that have distributed systems.

But its drawbacks led us to discard them quickly. The main is its high cost, both manufacturing and maintenance. The maintenance cost is due to the large of power amount that used. Another drawback of this type of system is its scalability, if you want to add a CPU in a supercomputer need software to change the whole system, and the same to remove a processing unit. Instead, in clusters just you need to modify the software on the master node and the node you want to add.

Once we have determined the different types of systems that perform simulations, we will explain how to carry this them out. To simulate multitasking of different users we need what is known as a process manager. This software has three main functions.

The queue manager is dedicated to sort tasks received by users to simulate. There are several ways to sort the different tasks depending you want to assign priorities. This type of tool has been created to solve the problems of Fairness and blocking generated by distributed computing. This mechanism slows processing tasks, but very convenient in some circumstances. In addition to the queue manager we can find the scheduler. This system is responsible for searching the resources requested by the user and assign tasks that were provided by it. Also keeps track of the task in process simulation and collects the results returned by the executor. The executor is responsible for simulating the task in the child nodes of the system and collecting the different states, but this is explained more precisely later.

13.4.2. Tools

After completing the contextualization of the project, the document describes various tools that we use for our project.

On the one hand, we will use the Apache Mesos program. This software is used to establish a hierarchy among the different CPUs that form our distributed system. This hierarchy is created to increase the ease of connection and better allocate resources. Apache Mesos consists of 3 types of nodes mainly.

The Zookeeper node is responsible for finding the master node within the distribution. When it finds the master node direction, tell to nodes slaves and schedulers this information, for use this to allocate tasks.

The Master node is the most important node in our system. This node is responsible for connecting nodes slaves or agents with the corresponding scheduler tasks. This node also serves as a bridge between communications. All messages pass through it. It also detects nodes fallen and errors in transmission.

Finally, we have the Agents nodes or Slaves nodes, these nodes are responsible for the processing of tasks. They communicate only with the master node. This sends tasks, process and send the result, and erasing the memory assigned for the task. This mechanism is explained more clearly as the document advances.

Another tool explained in this section are the various structures created to store the data that the user provides. These are the processes and tasks.

Process is the structure wider data we have. They have got the process ID, the resources needed to execute our jobs and command execution of these.

On the other hand, they are our tasks are the smallest units. In each one runs a job saved, the resources you need it, the identifier of the task and the node that it is sent for processing. These data are absolutely necessary to monitor the work and to run normally in the slave nodes.

In addition to the architecture of our distributed system and the type of data we use, other tools we use is the programming language. For this project we have opted for Python, due to its ease of learning and implementation. It is convenient in this case because it has OOP what makes us easier to create data types described above.

As for the GUI we use to make our simulations, we have opted for the use of I-Python Books. This interface is highly modular and can simulate the code parts separately in the same script. This series of features make it convenient for our type of simulations.

13.4.3. Development solution

To achieve our process manager must make the necessary code to perform the two parts. This project will only be undertaken by the scheduler using a single queue manager without any priority.

This leads to the description of our scheduler. To sending tasks to the node agents described above, we connect with our distributed system.

The first step is to find the master node of our system. To do this we connect the node making the request to Zookeeper who is the master node of the system. This node performs in turn a message of question for each of the master nodes containing wish list. The active master node sends a reply with its IP address. Once you have received this message, the Zookeeper sends the information to the scheduler who asked the question, and save on your list which is the active master node.

Once the Scheduler has the address of the master node makes a connection request with authentication data in the system.

When you have successfully authenticated, it performs resource request to the master node, of the tasks provided by the user.

When the master node receives the resource request message, recursively sends messages to the agent nodes or slaves asking how many resources are available in they. Each agent node returns this information to our master node. The master collects all messages and sends them to the scheduler to distribute tasks, you have to simulate in its active process nodes with the resources available.

For the simulation of tasks, the scheduler assigns each task to a node and sends this to the master node to distribute them among the selected slave nodes. When tasks are distributed correctly each agent node, receives a reserve the necessary resources to implement it and begins processing the task. Once completed the task within the slave node sends the completion status of the task to the master node and delete it of your memory. This, in turn, sends this information to the corresponding planner. When the scheduler receives the status of all tasks, it completes the process and stored in a script.txt for they can be viewed later. When all processes have terminated, ends the connection to the system and close the data file.

Throughout the document, we have been determining various functions of scheduler. But these data are not sufficient for our project. And we need to implement some basic functions for our system to connect to the executors mentioned above.

The main functions of this software are as follows. First, the scheduler receives the process information. It takes the resources necessary to perform the tasks contained in the process. And it makes the request to the master node as

explained above. Once data have become available nodes creates executable tasks for each element of the process, assigned resources and the necessary data from each of them. When you have created each sends through the master node to node to take the slave selected. The executor is responsible for the processing of the assignment and internal parallelization of it, so that the simulation be faster. In our case tracking task is superfluous, because we use an executor created by default in Mesos architecture. When the task is finished executing collects the various messages and sends them to master, this node returns the messages to the planner. When all tasks have completed the planning process closes.

13.4.4. End solutions

As the various tests that were conducted to see if the system works properly have been conducted mainly 3 types of tests. The first were aimed at checking the connection. simple messages were sent to the scheduler to simulate in the cluster in this test. The results of these simulations were satisfactory and the various types of tasks are simulated correctly.

The second tests were focused on checking that resource stocks were made correctly. In addition to displaying the graphical interface of our farm computer resources reserved coincided with requests by the user, we performed the simulation of the same task with different memory capacities and CPU seeing processing times varied proportionally.

The latest tests have been conducted to verify that the error handling. Our error handler is to complete all tasks of the process when one is wrong and end the connection. This system has been implemented by its simplicity and high efficiency in our simulation environment.

13.4.5. Fundamental conclusions

As for the completion of our project, we can conclude that both the library to create, as the user interface were successful. This is in part occurred for the decisions explained before in this document.

The use API Apache Mesos helped with issues of connection with our distributed system and to make the better distribution of executable elements in different sub-processes or tasks.

They have met the requirements required compatibility with existing systems in the system that we have and minimization of changes in structure.

13.4.6. Future works

Future improvements that can be considered are different due to the simplicity of the system.

One of them, is the use of an own executor of tasks, in this work is used the generated by default in Apache Mesos. This would give us more control in the tasks states within the cluster.

Another improvement that can be implemented in the future is a queue manager to determine the priorities of connecting multiple users in the cluster. Today is not necessary because the existing actual blocking rate is very low, and the complexity of the problem is too broad.